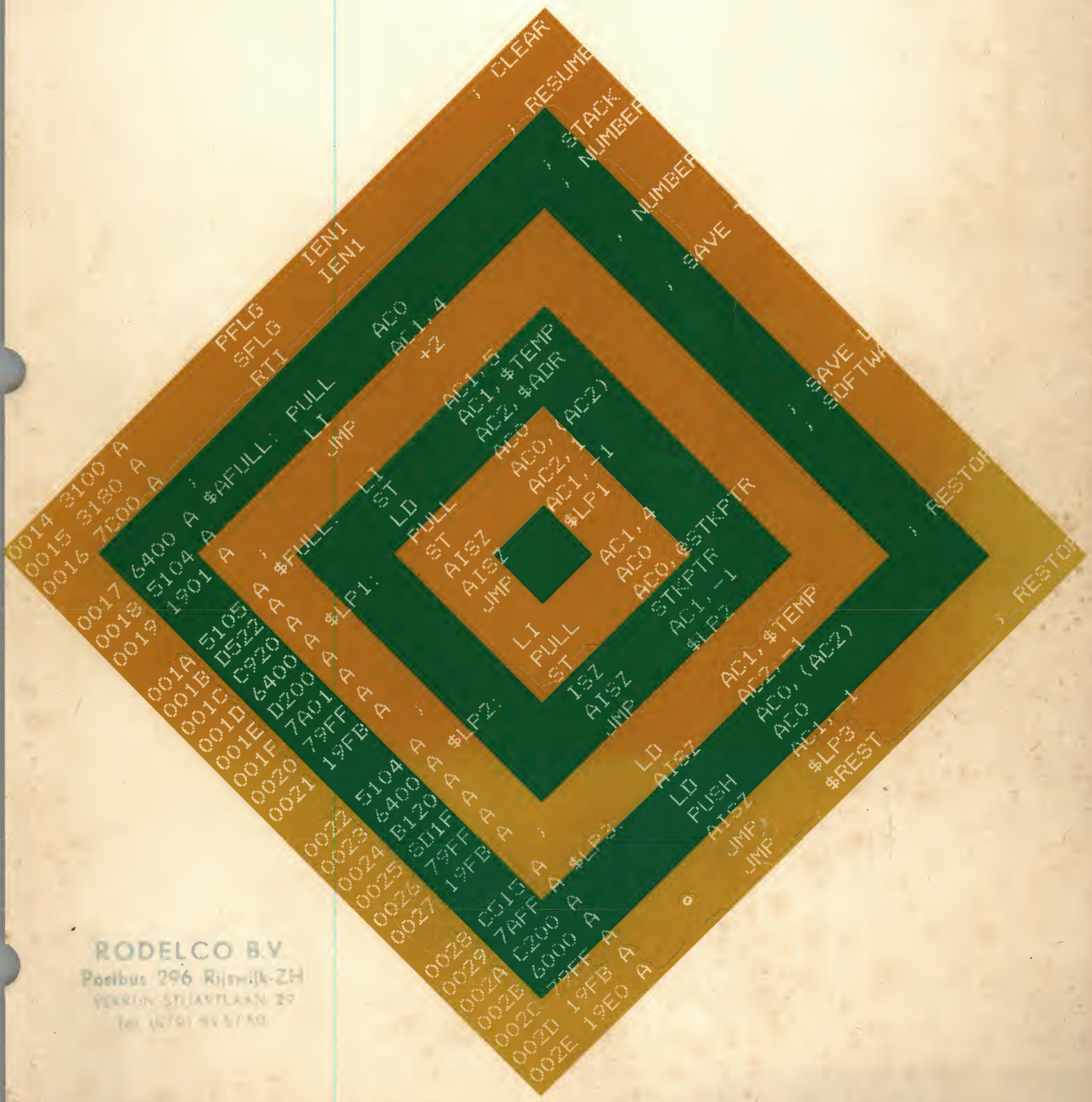
PACE

Microprocessor Assembly Language Programming Manual

National Semiconductor



Publication Number 4200130A
Order Number IPC-16S/969Y
January 1977

PACE
Processing And
Control Element

Assembly Language
Programming Manual

2900 Semiconductor Drive
Santa Clara, California 95051

PREFACE

The PACE Assembly Language Programming Manual provides tutorial and reference information required for writing user application programs. Component data sheets and information pertaining to prototyping systems are available in other publications.

The material in this publication is subject to change without notice. Changes will be reported in COMPUTE, the Microprocessor Users Group newsletter.

Copies of this publication and other National Semiconductor publications may be obtained from the National Semiconductor sales office or distributor serving your locality.

- PACE System Design Manual, Order Number IPC-16A/928.
- PACE (FORTRAN) Cross Assembler Software Package, Installation and Operating Instructions, Publication Number 4200073.
- PACE Conversational Assembler Users Manual, Publication Number 4200112.
- IPC-16A/520D MOS/LSI Single-Chip 16-bit Microprocessor (PACE) Data Sheet.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF ILLUSTRATIONS

LIST OF TABLES

Chapter 1

GENERAL INFORMATION

This publication provides information on how to write assembly language or machine language programs for a PACE microprocessor. Also contained is a brief description of the PACE microprocessor. A detailed description of the PACE microprocessor is contained in the PACE System Design Manual and the PACE Data Sheet. The user should consult the users or installation manual for the PACE assembler that he is using for instructions on how to assemble a PACE assembly language source program.

All publications referenced in this manual are listed in the preface.

The following is a brief description of the contents of chapter 2 through 8 of this manual.

Chapter 2, PACE Microprocessor Overview, describes the hardware functions that affect the programming of the PACE. Discussed are the registers, stack, input/output facilities, the interrupt system, a summary of the instruction set, addressing and other miscellaneous functions.

Chapter 3, Assembly Language, describes the coding conventions, elements, and structure of the PACE assembly language.

Chapter 4, Instruction Set, is a detailed description of the characteristics and the operation of the PACE instruction set.

Chapter 5, Assembler-Dependent Statements, describes three types of assembler-dependent statements: the comment, the assignment, and the directives.

Chapter 6, Macros, defines macro instructions, what they are, how to use them, and how to write them.

Chapter 7, Programming Techniques, provides programming examples that show how to write efficient code, link to subroutines, and perform input/output operations (programmed and interrupt).

Chapter 8, Assembler Input/Output Formats, describes the input/output formats that are common to all PACE assemblers.

Chapter 2

PACE MICROPROCESSOR OVERVIEW

This chapter describes the main features of the PACE microprocessor. Only those features with which the programmer is primarily concerned are discussed.

2.1 OPERATIONAL FEATURES

PACE is a programmable 16-bit parallel microprocessor. Functionally, PACE has a bidirectional data bus connecting the CPU, memory, and peripheral devices. Peripheral devices are assigned memory addresses, and any standard memory-reference instruction can be used for input/output operations.

Other important features are a 6-level priority interrupt system and the ability to process 8-bit or 16-bit data. Table 2-1 lists the operational features of PACE.

Table 2-1. Operational Features

Feature	Description
Instruction Word Length	16 bits
Data Word Length	8 bits or 16 bits
Instruction Set	46 Instructions
Arithmetic	Parallel, binary, fixed point, twos complement, 4-digit BCD addition
Memory	Up to 65,536 16-bit words of ROM or RAM
Registers	Four 16-bit general-purpose accumulators (two can be used as index registers)
	One 16-bit Status Register
	One 16-bit Program Counter (indirectly accessible)
Stack	10-word 16-bit Last-In/First-Out hardware stack
Addressing Modes	Direct addressing • Base page • Program counter relative (current page) • Indexed
	Indirect addressing • Base page • Program counter relative (current page) • Indexed
	Immediate addressing
Input/Output and Control	6 priority interrupt levels
	3 general-purpose jump-condition inputs
	4 general-purpose flag outputs
Typical Instruction-Execution Speed	12 microseconds

2.2 DATA REPRESENTATION

Data are represented internally in PACE in twos-complement integer notation. In twos-complement notation, the negative of a number is formed by complementing each bit in the data word and adding one to the complemented number. The sign of the number is indicated by the most significant bit (bit 15 for 16-bit data and bit 7 for 8-bit data). When the most significant bit is zero, the number is positive or zero; when it is one, the number is negative. Maximum range for a 16-bit number in this system is $7FFF_{16}$ ($+32767_{10}$) to 8000_{16} (-32768_{10}). Maximum range for an 8-bit number is $7F_{16}$ ($+127_{10}$) to 80_{16} (-128_{10}).

2.3 REGISTERS

The registers important to the PACE user are shown in figure 2-1 and discussed in the following paragraphs.

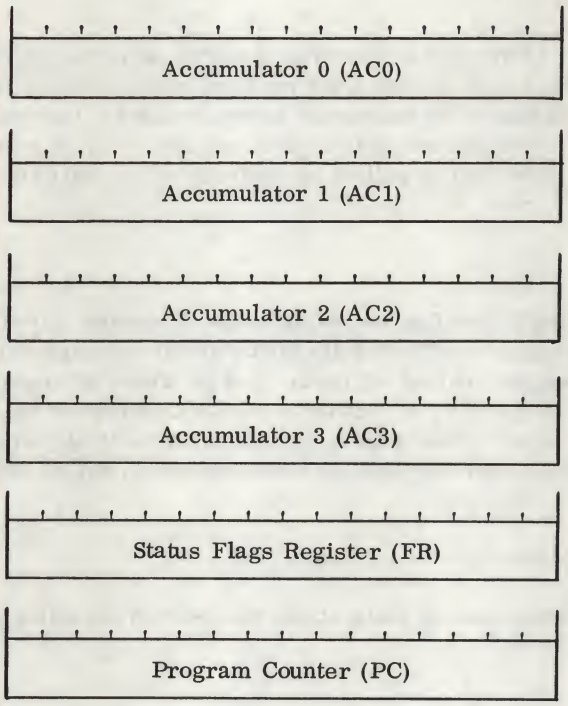


Figure 2-1. PACE Registers

2.3.1 Accumulators (AC0, AC1, AC2, AC3)

Four 16-bit accumulators (AC0, AC1, AC2, and AC3) are available to the PACE user. The accumulators are general-purpose registers used for performing arithmetic and logic operations, data transfers, skips, shifts, and rotates. The accumulators normally are used in the following capacities:

AC0 — primary data-handling register
AC1 — secondary data-handling register
AC2 — base register for indexed addressing of memory
AC3 — base register for indexed addressing of memory and peripherals

In assembly language, the accumulators are addressed by placing their numeric address (0, 1, 2, and 3 for AC0, AC1, AC2, and AC3, respectively) in the operand field of the source statement (see chapter 3, Assembly Language). The assignment statement (see 5.2) may be used to assign symbolic addresses to the accumulators.

2.3.2 Status Flags Register (FR)

The 16-bit Status Flags Register (FR) provides storage for interrupt, arithmetic, control, and status flags. FR automatically preserves system status. The user may operate on its contents as data, allowing masking, testing, and modification of several bit fields simultaneously. The bit position of each flag is shown below and defined in table 2-2.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Bit Position
'1'	F14	F13	F12	F11	BYTE	IEN	LINK	CY	OV	IE5	IE4	IE3	IE2	IE1	'1'	Flag

2.3.3 Program Counter (PC)

The Program Counter (PC) is a 16-bit register used by PACE to store the address of the next instruction to be executed. PC is incremented by 1 immediately after each instruction is fetched; therefore, the address of PC is always 1 greater than the address of the instruction being executed. The contents of PC are not directly accessible to the user; however, the contents may be obtained indirectly by executing a subroutine jump (JSR) to the next location in memory, and then by pulling the address off the top of the stack.

2.4 STACK (STK)

The hardware Stack (STK) in PACE provides ten 16-bit words of storage that are accessed sequentially in a last-in/first-out basis. The stack is used primarily for temporary storage of the contents of the Program Counter and the Status Flags Register during subroutine and interrupt service routine execution. A stack-full/stack-empty interrupt (level 1) is provided to facilitate off-stack storage in applications where additional stack capacity is desirable. The stack interrupt is generated automatically when the ninth word of the stack is filled, or whenever the stack becomes empty. For information on how to use the stack, see 7.1.

2.5 INSTRUCTION SUMMARY

The instruction set for PACE comprises 46 instructions divided into the following eight classes:

- Branch
- Skip
- Memory Data Transfer
- Memory Data Operate
- Register Data Transfer
- Register Data Operate
- Shift and Rotate
- Miscellaneous

The branch instructions provide the means to transfer control to any location in memory. Conditional branches are effected by using the BOC instruction, which allows testing of any one of 16 conditions, including status flags, the contents of AC0, and users inputs to the PACE microprocessor. (See table 4-1 for the condition codes.)

Additional testing capability is provided by the skip instructions, which provide memory or peripheral to register comparisons without altering data.

Table 2-2. Descriptions of Status and Control Flags

Register Bit	Flag Name	Description	Flag Code (fc)
0	'1'	Bit 0 is not used and is always in logic '1' state. Referencing bit 0 with SFLG or PFLG Instruction has no affect. (May be used as NOP Instruction.)	0000
1	IE1	Interrupt Enable level 1 (Stack full/empty interrupt)	0001
2 3 4 5	IE2 IE3 IE4 IE5	Interrupt Enable levels 2 through 5 are user interrupts. If Interrupt Enable (IEN) is a '1', and an interrupt request occurs, PACE executes the appropriate interrupt service routine. If IEN is a '0', the interrupt request for the level is ignored.	0010 0011 0100 0101
6	OV	Overflow Flag is set to state of twos-complement arithmetic overflow by arithmetic instructions. Overflow Flag is set high if sign bits (most significant bits) of two operands are identical and sign bit of result is different from sign bit of operands. If A, B, and R are sign bits of operands and result, then Overflow Flag is set according to equation $OV = (\overline{A} \bullet \overline{B} \bullet R) + (A \bullet B \bullet \overline{R})$ Sign bit is most significant bit for data length selected; thus, if data length is 8 bits, then bit 7 is sign bit; if data length is 16, then bit 15 is sign bit. State of OV Flag is affected by instructions ADD, DECA, SUBB, RADD, and RADC.	0110
7	CY	Carry Flag is set to state of binary or decimal carry output of adder by arithmetic instructions. Carry output is derived from most significant bit for data length specified by BYTE Flag. State of CY Flag is affected by instructions ADD, DECA, SUBB, RADD, and RADC.	0111
8	LINK	Link Flag is included in shift and rotate operations as specified by Shift and Rotate Instructions. Link Flag is unaffected if not selected.	1000
9	IEN	Master Interrupt Enable Flag simultaneously inhibits all five of lowest priority interrupt levels. No Interrupt Request is serviced unless individual Interrupt Enable Flag for associated Interrupt Request and master Interrupt Enable Flag are high. IEN Flag is set low every time any interrupt (except Level 0) is serviced. IEN Flag is set high by execution of Return from Interrupt Instruction (RTI).	1001
10	BYTE	BYTE Flag selects 8-bit data length when high and 16-bit data length when low.	1010
11 12 13 14	F11 F12 F13 F14	Flags 11 through 14 are general-purpose control flags. Flags 11 through 14 drive PACE output pins and may be used to directly control system functions (see PACE System Design Manual for details.	1011 1100 1101 1110
15	'1'	Bit 15 is not functional and is always in logic '1' state. Addressing bit 15 with SFLG or PFLG instruction sets the Level-0 Interrupt Enable.	1111

The memory data transfer instructions provide data transfers between the accumulators and memory or peripheral devices. The load with sign extended is provided to convert 8-bit, twos-complement data to 16-bit data, allowing 16-bit address modification when the 8-bit data length has been selected.

The memory data operate instructions provide operations between the principal working register (AC0) and memory or peripheral data. This includes both binary and BCD arithmetic instructions. The register data transfer instructions provide a complete set of transfer possibilities between the accumulators, flag register, and stack — and include the capability to load immediate data.

Register data operate instructions provide logical and arithmetic operations between any two accumulators. They may be used for address and data modification to reduce the number of (time-consuming) memory references in a program.

The shift and rotate instructions allow eight different operations that are useful for multiply, divide, bit scanning, and serial input/output operations.

The miscellaneous instructions include the capability to set or reset any of the 16 bits of the Status Flags Register individually.

The PACE instruction set is summarized in table 2-3. Refer to table 2-4 for definitions of the symbols used in the notation for describing the PACE instruction set. Upper-case mnemonics refer to units designated by fields of the instruction words; lower-case mnemonics refer to the numerical values of the corresponding fields. For example, in ACdr, AC indicates one of the four accumulators and dr gives the numeric value that indicates a particular accumulator.

2.6 MEMORY ADDRESSING

Three methods of accessing data by an instruction are available: immediate, direct, and indirect addressing.

2.6.1 Immediate Addressing

A statement that contains the operand value as part of the instruction has an immediate address. Immediate addressing is limited to certain operation mnemonics. All immediate instructions are absolute since the operand value does not change when the program is relocated.

2.6.2 Direct Addressing

Direct addressing is the most useful method of addressing available to the PACE user. Its flexibility is due primarily to the fact that there are three modes of direct addressing: base-page, Program Counter-relative, and indexed. The addressing mode is specified by the xr field of the instruction as shown in figure 2-2.

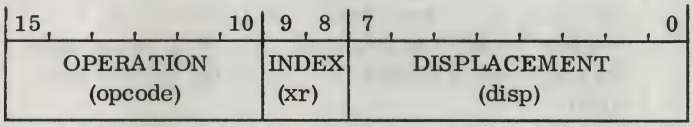


Figure 2-2. Memory-Reference Instruction Format

Table 2-3. PACE Instruction Summary

Description	Opcode Base	Source Statement	Instruction Format	Operation	Page
Branch Instructions					
Branch On Condition*	4000	BOC cc, address	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 0 1 0 0 | cc | disp	If cc true, (PC) ← (PC) + disp, B	
Jump	1800	JMP	0 0 0 1 1 0 | xr | disp	(PC) ← EA	
Jump Indirect	9800	JMP @	1 0 0 1 1 0 | xr | disp	(PC) ← (EA)	
Jump to Subroutine	1400	JSR address / disp(xr)	0 0 0 1 0 1 | xr | disp	(STK) ← (PC), (PC) ← EA	
Jump to Subroutine Indirect	9400	JSR @	1 0 0 1 0 1 | xr | disp	(STK) ← (PC), (PC) ← (EA)	
Return from Subroutine	8000	RTS	1 0 0 0 0 0 0 0	(PC) ← (STK) + disp	
Return from Interrupt	7C00	RTI disp	0 1 1 1 1 1 0 0 | disp	(PC) ← (STK) + disp, IEN = 1	
Skip Instructions					
Skip if Not Equal	F000	SKNE r,	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 1 1 1 1 | r	If (ACr) ≠ (EA), (PC) ← (PC) + 1, B	
Skip if Greater	9C00	SKG 0,	1 0 0 1 1 1	If (AC0) > (EA), (PC) ← (PC) + 1, B	
Skip if AND is Zero	B800	SKAZ 0, address / disp(xr)	1 0 1 1 1 0 | xr | disp	If [(AC0) ∧ (EA)] = 0, (PC) ← (PC) + 1, B	
Increment and Skip if Zero	8C00	ISZ	1 0 0 0 1 1	(EA) ← (EA) + 1; If (EA) = 0, (PC) ← (PC) + 1, B	
Decrement and Skip if Zero	AC00	DSZ	1 0 1 0 1 1	(EA) ← (EA) - 1; If (EA) = 0, (PC) ← (PC) + 1, B	
Add Immediate, Skip if Zero	7800	AISZ r, data	0 1 1 1 1 0 | r | data	(ACr) ← (ACr) + data; If (ACr) = 0, (PC) ← (PC) + 1	
Memory Data-Transfer Instructions					
Load	C000	LD r,	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 1 1 0 0 | r	(ACr) ← (EA), EA = (ACxr) + disp	
Load Indirect	A000	LD 0,@	1 0 1 0 0 0	(AC0) ← (EA), EA = ((ACxr) + disp)	
Store	D000	ST r, address / disp(xr)	1 1 0 1 | r | xr | disp	(EA) ← (ACr), EA = (ACxr) + disp	
Store Indirect	B000	ST 0,@	1 0 1 1 0 0	(EA) ← (AC0), EA = ((ACxr) + disp)	
Load with Sign Extended	BC00	LSEX 0,	1 0 1 1 1 1	(AC0) ← (EA) bit 7 extended	
Memory Data Operate Instructions					
AND	A800	AND 0,	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 1 0 1 0 1 0	(AC0) ← (AC0) ∧ (EA)	
OR	A400	OR 0,	1 0 1 0 0 1	(AC0) ← (AC0) ∨ (EA)	
Add	E000	ADD r, address / disp(xr)	1 1 1 0 | r | xr | disp	(ACr) ← (ACr) + (EA); CY, OV	
Subtract with Borrow	9000	SUBB 0,	1 0 0 1 0 0	(AC0) ← (AC0) + ~(EA) + CY; CY, OV	
Decimal Add	8800	DECA 0,	1 0 0 0 1 0	$(AC0) \leftarrow (AC0)_{10} + (EA)_{10} + CY$; CY, OV	

* The Branch On Condition Instruction Condition Codes (cc) are given in table 4-1.

NOTE: The operation equations are read from left to right.

Table 2-3. PACE Instruction Summary (Continued)

Description	Opcode Base	Source Statement	Instruction Format	Operation	Page
Register Data-Transfer Instructions			15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0		
Load Immediate	5000	LI r, data	0 1 0 1 0 0 | r | data	(ACr) ← data (bit 7 extended)	
Register Copy	5C00	RCPY sr, dr	0 1 0 1 1 1 | dr | sr | 0 0 0 0 0 0	(ACdr) ← (ACsr)	
Register Exchange	6C00	RXCH sr, dr	0 1 1 0 1 1 | dr | sr | 0 0 0 0 0 0	(ACdr) ↔ (ACsr)	
Exchange Register and Stack	1C00	XCHRS r	0 0 0 1 1 1 | r | 0 0 0 0 0 0 0 0	(ACr) ↔ (STK)	
Copy Flags into Register	0400	CFR r	0 0 0 0 0 1 | r | 0 0 0 0 0 0 0 0	(ACr) ← (FR)	
Copy Register into Flags	0800	CRF r	0 0 0 0 1 0 | r | 0 0 0 0 0 0 0 0	(FR) ← (ACr)	
Push Register onto Stack	6000	PUSH r	0 1 1 0 0 0 | r | 0 0 0 0 0 0 0 0	(STK) ← (ACr)	
Pull Stack into Register	6400	PULL r	0 1 1 0 0 1 | r | 0 0 0 0 0 0 0 0	(ACr) ← (STK)	
Push Flags onto Stack	0C00	PUSHF	0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0	(STK) ← (FR)	
Pull Stack into Flags	1000	PULLF	0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0	(FR) ← (STK)	
Register Data-Operate Instructions			15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0		
Register Add	6800	RADD sr, dr	0 1 1 0 1 0 | dr | sr | 0 0 0 0 0 0	(ACdr) ← (ACdr) + (ACsr); CY, OV	
Register Add with Carry	7400	RADC sr, dr	0 1 1 1 0 1 | dr | sr | 0 0 0 0 0 0	(ACdr) ← (ACdr) + (ACsr) + CY; CY, OV	
Register AND	5400	RAND sr, dr	0 1 0 1 0 1 | dr | sr | 0 0 0 0 0 0	(ACdr) ← (ACdr) ∧ (ACsr)	
Register Exclusive-OR	5800	RXOR sr, dr	0 1 0 1 1 0 | dr | sr | 0 0 0 0 0 0	(ACdr) ← (ACdr) ∇ (ACsr)	
Complement and Add Immediate	7000	CAI r, data	0 1 1 1 0 0 | r | data	(ACr) ← ~ (ACr) + data (bit 7 extended)	
Shift and Rotate Instructions			15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0		
Shift Left	2800	SHL r, n, ℓ	0 0 1 0 1 0 | r | n | ℓ	(ACr) ← (ACr) shifted left n places, w/wo LINK; B	
Shift Right	2C00	SHR r, n, ℓ	0 0 1 0 1 1 | r | n | ℓ	(ACr) ← (ACr) shifted right n places, w/wo LINK; B	
Rotate Left	2000	ROL r, n, ℓ	0 0 1 0 0 0 | r | n | ℓ	(ACr) ← (ACr) rotated left n places, w/wo LINK; B	
Rotate Right	2400	ROR r, n, ℓ	0 0 1 0 0 1 | r | n | ℓ	(ACr) ← (ACr) rotated right n places, w/wo LINK; B	
Miscellaneous Instructions			15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0		
Halt	0000	HALT	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	Halt	
Set Flag	3080	SFLG fc*	0 0 1 1 | fc | 0 0 0 0 0 0 0 0	$(FR_{fc}) \leftarrow 1$	
Pulse Flag	3000	PFLG fc*	0 0 1 1 | fc | 0 0 0 0 0 0 0 0	$(FR_{fc}) \leftarrow 1$, $(FR_{fc}) \leftarrow 0$	
No Operation	5C00	NOP	0 1 0 1 1 1 0 0 0 0 0 0 0 0 0 0	(PC) ← (PC) + 1	

* The flags (fc) in the Status Flags Register (FR) are given in table 2-2.

NOTE: The operation equations are read from left to right.

Table 2-4. Symbols and Notations

Symbol and Notation	Meaning
AC	Accumulator (AC0, AC1, AC2 or AC3).
address	Symbol representing a memory location.
B	Indicates instruction execution is affected by the state of the BYTE Flag in FR.
BYTE	The BYTE Flag in FR.
cc	Condition code. A 4-bit value used in conditional-branch instructions.
CY	Carry Flag. Indicates the Carry Flag in FR is set or cleared in the instruction.
data	8-bit immediate data field.
disp	Displacement. An 8-bit address modified in memory-reference and branch instructions. Disp is a signed twos-complement number except when nonsplit base page is referenced; in that case, disp is unsigned.
dr	Destination register. Denotes number of accumulator (AC0, AC1, AC2, or AC3).
EA	Effective Address. The address actually used in the execution of an instruction.
fc	Denotes the number of a flag (15 to 0) in FR.
FR	Status Flags Register.
IEN	Interrupt Enable Flag in FR.
ℓ	The link indicator in a shift or rotate instruction. If ℓ equals 1, the LINK Flag in FR is included in the shift or the rotate instruction.
LINK	LINK Flag in FR.
n	Unsigned number indicating the number of bit positions to be shifted or rotated.
OV	Overflow Flag in the Status Flag Register.
PC	Program Counter. During address formation, PC is incremented by 1 to contain an address 1 greater than the address of the instruction being executed.
r	Number of accumulator specified in instruction format.
STK	Top word of 10-word last-in/first-out stack.
sr	Source register. Denotes number of Accumulator (AC0, AC1, AC2, or AC3).
xr	Denotes the addressing mode: base-page-relative if xr = 00, PC-relative if xr = 01 or AC2- or AC3-relative (indexed) if xr = 10 or 11.
( )	"Contents of." For example, (EA) is contents of the Effective Address.
[ ]	"Result of."
$\sim$	Ones complement of value immediately to right of $\sim$.
$\rightarrow$	"Replaces."
$\leftarrow$	"Is replaced by."
$\leftrightarrow$	"Exchange."
@	When used in the operand field of an instruction, indicates indirect addressing.
10^{+}	Modulo 10 addition.
$\wedge$	AND operation.
$\vee$	Inclusive-OR operation.
$\triangledown$	Exclusive-OR operation.

2.6.2.1 Base-Page Addressing

Memory in PACE is divided into two sections: base page (or base sector) and top sector. Base page is a section of memory that can be addressed from any location of memory. Top sector is any portion of memory that is not base page.

Base-page addressing is specified by the value 00 in the xr field of the instruction. Two types of base-page addressing are available. The type of base-page addressing selected is determined by the state of the Base Page Select (BPS) input signal on the PACE microprocessor (see PACE Data Sheet for details on BPS input signal). When BPS is low (0), the effective address for the instruction is formed by setting bits 15 through 8 to zero and using the 8-bit displacement (disp) field for bits 7 through 0. Thus, the first 256 words of memory (locations 0 to 255) can be addressed. When BPS is high (1), the effective address is formed by treating bit 7 as the sign bit and setting bits 15 through 8 equal to bit 7 (propagating the sign bit). Thus, the last 128 words of memory and the first 128 words of memory can be addressed (-128 to +127). The latter technique is useful for splitting the base-page between read/write memory and read-only memory or between memory and peripherals. Consequently, base-page addressing provides a convenient method of accessing common constants or variables or peripherals. Figure 2-3 illustrates base-page addressing.

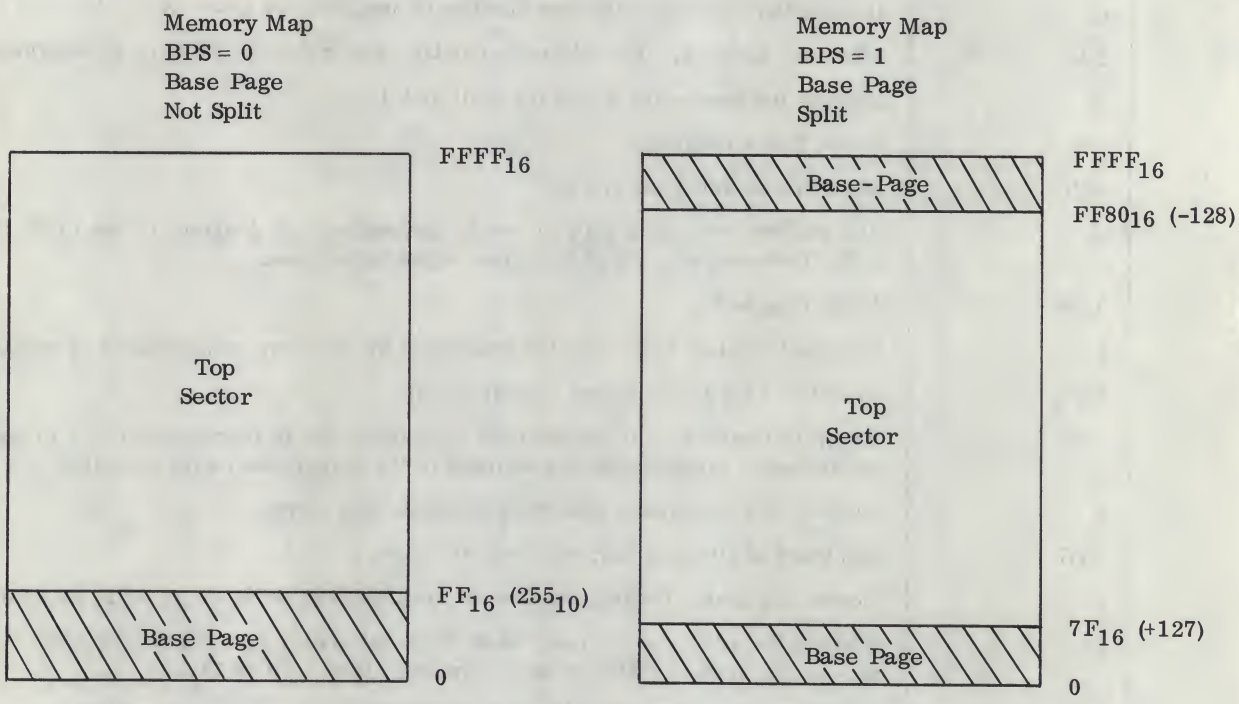


Figure 2-3. Base-Page Addressing

2.6.2.2 Program Counter-Relative Addressing

A Program Counter-relative address is formed by adding the current contents of the Program Counter (PC) to the value specified in the operand field. The value is treated as a signed number since its sign bit (bit 7) is propagated through bits 8 through 15. Thus, Program Counter-relative addressing is permitted from -128 to +127 locations from the PC value. However, at the time the Program Counter-relative address is calculated, the Program Counter already is incremented and is pointing to the next memory location. Therefore, the actual addressing range is -127 to +128 from the current instruction. Figure 2-4 illustrates PC-relative addressing.

NOTE

When using this addressing mode, the assembler program automatically assigns value 01 to the index register field in the machine instruction format, specifying the Program Counter-relative addressing.

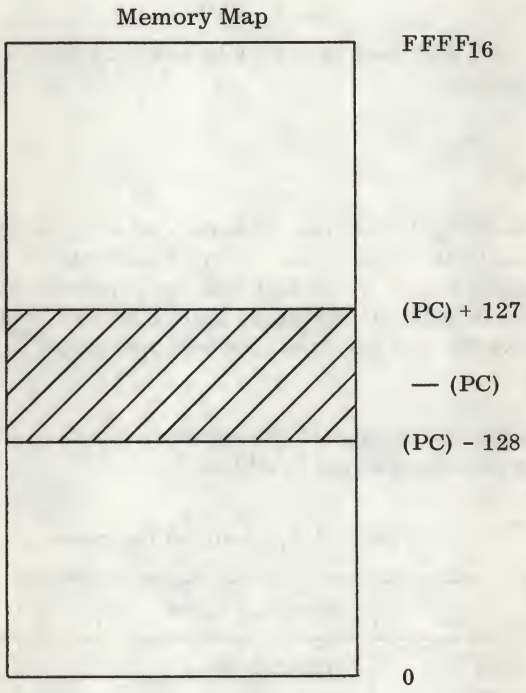


Figure 2-4. PC-Relative Addressing

Using assembly language, the programmer cannot explicitly specify Program Counter-relative addressing. If the programmer specifies a base-page reference without indexed addressing in the operand field, the assembler program automatically indicates the base-sector address mode. If a top-sector reference is indicated, the assembler program attempts Program Counter-relative addressing. If neither of these modes can be used and indirect addressing is available, the assembler will modify the instruction to make an indirect reference through a pointer.

If all modes fail, an addressing error occurs.

2.6.2.3 Indexed Addressing

Indexed addressing enables the programmer to address any location in memory by using a base-register addressing scheme. Base-register addressing requires the designation of an accumulator (containing a base address) and a displacement value for specifying a memory location. PACE adds the contents of the register to the number formed from the displacement value to yield the address. For example, assume that AC2 contains a base address of 300, and the displacement value is 120. The displacement is added to the base address and the result is an address of 420.

Only Accumulators 2 and 3 may be used as base (index) registers, and a base address must be previously assigned to the selected register before that register is used in an address.

PACE indexed addressing allows the programmer to address 256 words around the base address; that is, the base address represents the middle of a floating page. The referenced address can be -128 through +127 from the base.

An indexed address operand contains the displacement immediately followed by the index register address in parentheses. The register address must be absolute and evaluate to 2 or 3 (Accumulator 2 or 3). The displacement number is treated as a signed 8-bit number from -128 to +127.

NOTE

Only split base page may be indexed throughout its range.

2.6.3 Indirect Addressing

An indirect address operand specifies the address of a memory location that holds the address of the data to be used as the effective address by the instruction. Indirect addresses fall into three categories; base page, Program Counter-relative, and indexed. The initial address is calculated by using the same methods that are used for direct addresses. Indirect addressing is limited to certain operations and is specified by an @ before the displacement value in the operand field. Indirect addressing is used for long jumps to any location in memory.

The instruction fields provide the data required to calculate an effective address at execution time. Details of the possible addressing modes are summarized in table 2-5.

Table 2-5. Address Operands

Type	Operand Field	Address Calculation
Direct base-page	displacement	EA = disp
Direct PC-relative		EA = disp + PC
Indirect base-page	@displacement	EA = (disp)
Indirect PC-relative		EA = (disp + PC)
Direct indexed	displacement (index)	EA = disp + ACxr
Indirect indexed	@displacement (index)	EA = (disp + ACxr)

NOTES:
- EA — effective address specified by the instruction. The contents of the effective address are used during execution of an instruction.
- disp — stands for displacement value and is an 8-bit, signed twos-complement number except when base-page address is specified.
- PC — Program Counter.
- ACxr — index register (AC2 or AC3).

2.6.4 Operator Address Classes

Instruction statement memory-reference operators are separated into three classes according to addressing capability. The classes are defined as follows:

Class 1 ADD, SUBB, SKG, SKNE, AND, OR, SKAZ, ISZ, DSZ, LSEX, DECA

a. May directly address all of base page.
b. May use indexed addressing with displacement range -128 through 127.
c. If not indexed, assembler program attempts Program Counter-relative addressing for top-sector reference.

Class 2 LD, ST, JMP, JSR

a. May directly address all of base page.
b. May use indexed addressing with displacement range -128 through 127.
c. If not indexed, assembler program attempts Program Counter-relative addressing for top-sector reference.
d. May use indirect addressing to address all of memory.

NOTE

Indirect addressing for LD and ST Instructions is restricted to using AC0 as the operand register.

e. For a top-sector reference, if the instruction is not already indexed or marked indirect, and if the use of Program Counter-relative addressing is not possible, the assembler forces indirect addressing through a pointer. Generation of a base-page pointer may be avoided by providing an explicit top-sector pointer (see 5.3.15) within addressing range or by specifying pools (see 5.3.16) of pointer space in top sector. If the .NOBAS directive (see 5.3.17) is used, assembler-generated base-page pointers are flagged with a warning error message.

Class 3 BOC Assembler attempts Program Counter-relative addressing.

2.7 INTERRUPT SYSTEM

The PACE microprocessor has a 6-level priority interrupt structure. Interrupt Level 0 (IR0) has the highest priority and Interrupt Level 5 (IR5) has the lowest priority. Each level is provided with an individual Interrupt Enable as shown in figure 2-5. A master Interrupt Enable (IEN) is provided to allow the five lower-priority levels (IR1 through IR5) to be enabled or disabled as a group. Negative true interrupt request inputs (NIR2 through NIR5) are provided to allow several interrupts to be wire-ORed to each input.

When an interrupt request occurs on the five lower-priority interrupts, the associated Interrupt Request Latch (IR1 through IR5) is set if the corresponding Interrupt Enable Input is true. Since the latch can be set by any pulse exceeding one clock period, narrow timing or control pulses can be captured. If the master Interrupt Enable (IEN) is set, then an interrupt is generated and acknowledged after completing execution of the current instruction.

During the interrupt sequence, an address is provided by the output of the priority encoder. This address is used to access memory locations 2 through 6 which contain the addresses of the user's interrupt service routines for interrupts 1 through 5, respectively (see table 2-6). Before executing the service routine for the interrupt, the Program Counter is pushed on the stack and IEN is cleared. The interrupt service routine may set IEN after turning off the Interrupt Enable Flag for the level currently being serviced (or resetting the interrupt request). The Interrupt Enable Flags may be set by the Set Flag (SFLG) Instruction and cleared by the Pulse Flag (PFLG) Instruction. If an Interrupt Enable Flag is set or cleared, one more instruction is executed before the interrupt is enabled or disabled. The Return from Interrupt Instruction (RTI) also may be used to set IEN high. In this case, there is no delay and a pending interrupt takes effect immediately after execution of RTI.

The non-maskable level 0 interrupt (IR0) is an exception to the interrupt procedure given on the preceding page. The level 0 interrupt is serviced by first setting the Level 0 Interrupt Enable (IR0 INT Enable in figure 2-5) low to lock out all other possible interrupts. Next the Program Counter is stored in a location specified by memory location 7 (see table 2-6). Then, the instruction at memory location 8 is executed. Storing the Program Counter contents in a memory location instead of on the Stack prevents generation of a Stack-full interrupt.

To return from a level 0 interrupt, the PFLG 15 or SFLG 15 instruction is executed to set the Level 0 Interrupt Enable Output high after execution of one additional instruction. The additional instruction is typically a JMP@ through memory location 7, which contains the Program Counter contents address. Interrupt Level 0 is typically used by the control panel, which can always interrupt the application program and does not affect system status. Level 0 interrupts are generated by exercising the NHALT (typically the HALT instruction) and CONTIN lines (see the PACE System Design Manual for details).

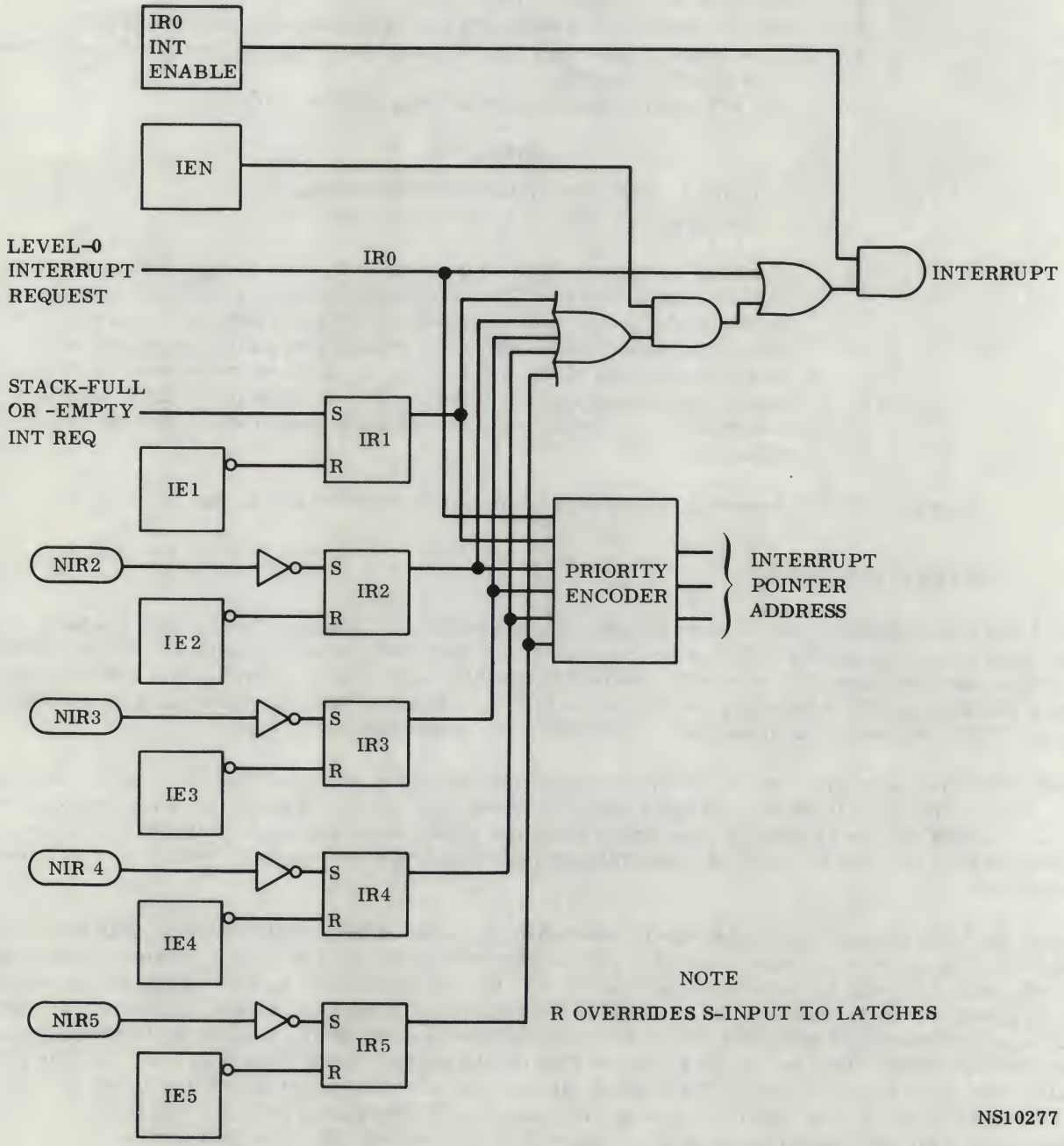


Figure 2-5. PACE Interrupt System

Table 2-6. Locations of Interrupt Pointers

Memory Location	Function
2	Interrupt 1 Service Routine Address Pointer
3	Interrupt 2 Service Routine Address Pointer
4	Interrupt 3 Service Routine Address Pointer
5	Interrupt 4 Service Routine Address Pointer
6	Interrupt 5 Service Routine Address Pointer
7	Interrupt 0 Program Counter Save Address Pointer
8	Interrupt 0 Service Routine
•	•
•	•
•	•

2.8 DATA INPUT/OUTPUT

All data transfers between the PACE CPU and external memory or peripherals takes place over 16 data lines. Peripheral devices are assigned memory addresses, so any memory-reference instruction can be used for input/output operations. Because of variations in peripheral devices, depending on the function performed, a standard input/output operation cannot be described here. See the PACE System Design Manual and the PACE Data Sheet for details on input/output operations.

2.9 8-BIT DATA LENGTH

In applications where the principal data length is 8 bits, using an 8-bit data memory and taking advantage of the data-length-selection hardware features may be desirable. The data-length input modifies the operation of shift instructions and status flags to handle 8-bit data. The instruction memory always is 16 bits wide, and proper execution of the 16-bit instructions occurs independently of the data length selected. The use of 16-bit instructions in 8-bit data applications provides higher execution speeds. The hardware design allows the system to be used in the 8-bit mode for variable data, while, at the same time, using all 16 bits of the PACE registers and stack to manipulate 16-bit memory addresses. Thus, the 16-bit instruction set is used to manipulate 8-bit data.

NOTE

The use of a status flag to specify data length enables the microprocessor to be switched between 8- and 16-bit modes under program control.

For information on using 8-bit data length see 7.4.

Chapter 3

ASSEMBLY LANGUAGE

3.1 INTRODUCTION

A program is a list of instructions in a specific sequence defined by the programmer to operate on data. An instruction is a statement that contains two basic parts: an operation code defining the operation to be performed and one or more operands defining the location of the data or specifying a device to be used.

The sequence of instructions in the program performs the following functions:

- Establishes working areas (areas to which data may be moved for manipulation) in storage.
- Specifies constants (such as values used in arithmetic calculations or symbols used to set switches).
- Specifies the appropriate operations to move data, perform appropriate tests and calculations, handle exceptional conditions, and arrange data in appropriate output formats.

Many programmers find a flowchart assists in coding instruction statements. A process flowchart contains all information required by a programmer to write a usable program. Usually, each symbol of a flowchart represents several statements to be coded. Some of the symbols represent data manipulation activities. Others represent operations required by the processor. Hosekeeping activities, such as setting counters or clearing output areas, are typical examples of processor operations.

3.1.1 Assembly Language

Assembly language is a machine-oriented symbolic programming language that allows the programmer to specify operations and operands with symbolic notations instead of binary notation. The programmer specifies alphabetic or alphanumeric symbols in place of memory addresses for data and instructions. In addition, assembly language provides mnemonic operation codes. For example, for a jump to the address labeled LOOP, could be coded in binary notation as

```
0001 1000 1111 0101
```

or in hexadecimal notation as

```
1  8  F  5
```

or in assembly language notation as

```
JMP  LOOP
```

JMP is the symbolic representation of the Jump Instruction, and LOOP is the symbolic representation of the destination address.

During the hexadecimal coding process, the programmer is responsible for counting address locations and determining if the jump (in the example) is best executed through the base page, or is performed as a displacement from an indexed value or as a displacement from the present memory position. With assembly-language coding, if the destination address of the jump is labeled LOOP, the assembler determines the appropriate addressing mode and displacement, thus relieving the programmer from performing the task. The assembler program keeps track of the relationship between memory addresses and symbolic terms. Thus, the assembler program can substitute the appropriate address for the symbolic name in the object code version of the instruction generated by the assembler. The processor executes the generated instruction.

In summary, using a symbolic language rather than binary or hexadecimal notation for programming provides the user with the following important advantages as follows:

- Mnemonic operation codes can be used to designate an operation.
- Data and instruction addresses can be assigned symbolic names for use in subsequent instructions.
- The ultimate memory location of the program is handled by the assembler or loader.
- The programmer may specify constant data in alphabetic, hexadecimal, or decimal format rather than binary format.
- Symbolic programs are easily modified because additional statements may be inserted into an existing statement sequence without concern for changing addresses in the existing instructions.

3.1.2 Assembler Programs

The statements written in symbolic assembler language must be translated into machine language before the processor can execute the instructions.

The conversion of the program from a symbolic representation to binary representation is performed by the assembler program. The assembler program translates the symbolic mnemonics into a binary machine-language program. This conversion is called the assembly process.

The assembly process starts with the symbolic source program written by the programmer. An example of a source program is shown in figure 3-1. The statements shown may be punched on paper tape or cards for input to one of the assembler programs.

The source program contains two basic types of statements: symbolic machine instructions and assembler-dependent statements, such as directives.

Machine-language instructions are used to request the processor to perform a sequence of operations during program execution time. Assembly-language instruction statements are a one-for-one symbolic representation of actual binary machine-language instructions. For each assembly-language instruction statement, the assembler program generates an equivalent machine-language instruction in the object program.

Operands of machine-language instructions represent storage locations, registers, immediate data, or constant values. A machine-language instruction statement may be identified by assigning a name (label) to it. The value of the label is the address of the assembled machine-language instruction.

Directives are used to request the assembler program to perform certain operations during the assembly process. The requested operations include assisting the programmer in data and symbol definition, checking and documenting the program, controlling the assignment of storage addresses, program sectioning and linking, defining data and storage fields, and controlling the assembler auxiliary functions to be performed by the assembler program. With few exceptions, directives do not result in the generation of any machine-language code in the object program.

Operands of directives provide the information needed by the assembler program to perform the designated operation.

Two outputs are generated as a result of running a source program (programmer-generated statements) through the assembler program: (1) a load module punched on paper tape or cards, consisting of actual machine-language instructions corresponding to the source program statements, and (2) a program listing showing source statements side by side with the object code instructions created from the statements. Most programmers work with the program listing once it is available. An example of a PACE program listing is shown in figure 3-2.

CODING FORM

PROGRAM EXAMPLE PROGRAM | PUNCHING INSTRUCTIONS | GRAPHIC PUNCH | PAGE 1 OF 3
PROGRAMMER J. CODER | DATE

STATEMENT

LABEL	OPERATION	OPERAND	COMMENTS	IDENTIFICATION
	.TITLE	SSORT, SIMPLE SORT (08/01/73)		
	.GLOBL	SSORT	;CAN REFERENCE ENTRY POINT FROM	
			;SEPARATE ASSEMBLY	
;	SSORT SORTS A VECTOR OF		SINGLE-WORD CONSTANTS INTO ASCENDING	
;	ORDER. CALLING SEQUENCE		IS:	
;				
;	JSR	SSORT	;CALL	
;	.WORD	VECTOR	;ADDRESS OF VECTOR	
;	.WORD	VECTOR+LENGTH-1	;ADDRESS OF LAST WORD OF VECTOR	
;	...		;NORMAL RETURN	
;				
FLAG:	.WORD	0	;IF NON-ZERO, SWAP MADE DURING PASS	
TAB:	.=.+1		;VECTOR ADDRESS	
TABEND:	.=.+1		;SORT LIMIT	
REGS:	.=.+4		;REGISTER SAVE AREA	
SSORT:	ST	0, REGS	;SAVE REGISTERS	
	ST	1, REGS+1		
	ST	2, REGS+2		
	ST	3, REGS+3		
	PULL	2	;OBTAIN ADDRESS OF PARAMETER LIST	
	PUSH	2	;BY PULLING PC FROM STACK TO AC2	
	LI	0, 0		
	ST	0, FLAG	;CLEAR FLAG	
	LD	3, 1(2)	;END OF VECTOR ADDRESS TO AC3	
	AISZ	3, -1	;DECREMENT ADDRESS	

CODING FORM

PROGRAM EXAMPLE PROGRAM | PUNCHING INSTRUCTIONS | GRAPHIC PUNCH | PAGE 2 OF 3
PROGRAMMER J. CODER | DATE

STATEMENT

LABEL	OPERATION	OPERAND	COMMENTS	IDENTIFICATION
	ST	3, TABEND		
	LD	3, (2)	;VECTOR ADDRESS	
	ST	3, TAB		
LOOP:	LD	0, (3)	;GET A VALUE	
	SKG	0, 1(3)	;COMPARE AGAINST NEXT VALUE	
	JMP	TEST	;VALUES IN ORDER	
	LD	1, 1(3)	;SWAP VALUES	
	ST	0, 1(3)		
	ST	1, 0(3)		
	LI	1, 1	;SET SORT FLG NON-ZERO	
	ST	1, FLAG		
TEST:	AISZ	3, 1	;INCREMENT TABLE POINTER	
	RCPY	3, 0		
	SKG	0, TABEND	;FINISHED THIS PASS?	
	JMP	LOOP	;NO	
	LD	0, FLAG	;YES - DID WE MAKE A SWAP?	
	AISZ	0, 0		
	JMP	.+2	;YES - CONTINUE	
	JMP	OUT	;NO - SORT DONE	
	LI	0, 0	;INITIALIZE FOR NEXT PASS	
	ST	0, FLAG		
	LD	3, TAB		
	JMP	LOOP		
OUT:	LD	0, REGS	;RESTORE REGISTERS	
	LD	1, REGS+1		

CODING FORM

PROGRAM EXAMPLE PROGRAM | PUNCHING INSTRUCTIONS | GRAPHIC PUNCH | PAGE 3 OF 3
PROGRAMMER J. CODER | DATE

STATEMENT

LABEL	OPERATION	OPERAND	COMMENTS	IDENTIFICATION
	LD	2, REGS+2		
	LD	3, REGS+3		
	RTS	2		
	.END			

NS10321

Figure 3-1. Example Source Program

```
PACE CROSS ASSEMBLER REV-A 10/21/74
SSORT SIMPLE SORT (08/01/73)


 1                          .TITLE  SSORT,'SIMPLE SORT (08/01/73)'
 2                          .GLOBL  SSORT             ;CAN REFERENCE ENTRY POINT F
 3                                                    ;SEPARATE ASSEMBLY
 4                  ;       SSORT SORTS A VECTOR OF SINGLE-WORD CONSTANTS INTO A
 5                  ;       ORDER.  CALLING SEQUENCE IS:
 6                  ;       JSR     SSORT             ;CALL
 7                  ;       .WORD   VECTOR            ;ADDRESS OF VECTOR
 8                  ;       .WORD   VECTOR+LENGTH-1   ;ADDRESS OF LAST WORD OF VEC
 9                  ;       ...                       ;NORMAL RETURN
10                  ;
11 0000 0000 A  FLAG:   .WORD   0                 ;IF NON-ZERO, SWAP MADE DURI
12       0002   TAB:    .=.+1                     ;VECTOR ADDRESS
13       0003   TABEND: .=.+1                     ;SORT LIMIT
14       0007   REGS:   .=.+4                     ;REGISTER SAVE AREA
15 0007 D1FB T  SSORT:  ST      0,REGS            ;SAVE REGISTERS
16 0008 D5FB T          ST      1,REGS+1
17 0009 D9FB T          ST      2,REGS+2
18 000A DDFB T          ST      3,REGS+3
19 000B 6600 A          PULL    2                 ;OBTAIN ADDRESS OF PARAMETER
20 000C 6200 A          PUSH    2                 ;BY PULLING PC FROM STACK TO
21 000D 5000 A          LI      0,0
22 000E D1F1 T          ST      0,FLAG            ;CLEAR FLAG
23 000F CE01 A          LD      3,1(2)            ;END OF VECTOR ADDRESS TO AC
24 0010 7BFF A          AISZ    3,-1              ;DECREMENT ADDRESS
25 0011 DDF0 T          ST      3,TABEND
26 0012 CE00 A          LD      3,(2)             ;VECTOR ADDRESS
27 0013 DDED T          ST      3,TAB
28 0014 C300 A  LOOP:   LD      0,(3)             ;GET A VALUE
29 0015 9F01 A          SKG     0,1(3)            ;COMPARE AGAINST NEXT VALUE-
30 0016 1905 T          JMP     TEST              ;VALUES IN ORDER
31 0017 C701 A          LD      1,1(3)            ;SWAP VALUES
32 0018 D301 A          ST      0,1(3)
33 0019 D700 A          ST      1,0(3)
34 001A 5101 A          LI      1,1               ;SET SORT FLAG NON-ZERO
35 001B D5E4 T          ST      1,FLAG
36 001C 7B01 A  TEST:   AISZ    3,1               ;INCREMENT TABLE POINTER
37 001D 5CC0 A          RCPY    3,0
38 001E 9DE3 T          SKG     0,TABEND          ;FINISHED THIS PASS?
39 001F 19F4 T          JMP     LOOP              ;NO
40 0020 C1DF T          LD      0,FLAG            ;YES - DID WE MAKE A SWAP?
41 0021 7800 A          AISZ    0,0
42 0022 1901 T          JMP     .+2               ;YES - CONTINUE
43 0023 1904 T          JMP     OUT               ;NO - SORT DONE
44 0024 5000 A          LI      0,0               ;INITIALIZE FOR NEXT PASS
45 0025 D1DA T          ST      0,FLAG
46 0026 CDDA T          LD      3,TAB
47 0027 19EC T          JMP     LOOP
48 0028 C1DA T  OUT:    LD      0,REGS            ;RESTORE REGISTERS
49 0029 C5DA T          LD      1,REGS+1
50 002A C9DA T          LD      2,REGS+2
51 002B CDDA T          LD      3,REGS+3
52 002C 8002 A          RTS     2
53       0000           .END

                                                                        NS10322
```

Figure 3-2. Example of PACE IMP-16 Cross Assembly Listing

As a source program is assembled, an analysis for errors in the use of the assembly language is performed. Any detected errors are indicated on the program listing to assist the programmer in debugging.

The flowchart in figure 3-3 shows the relationship of the assembler program to the programming process.

In summary, a program goes through six basic processes: (1) design, (2) coding of source statements, (3) assembly run, (4) debugging, (5) test run, and (6) production run.

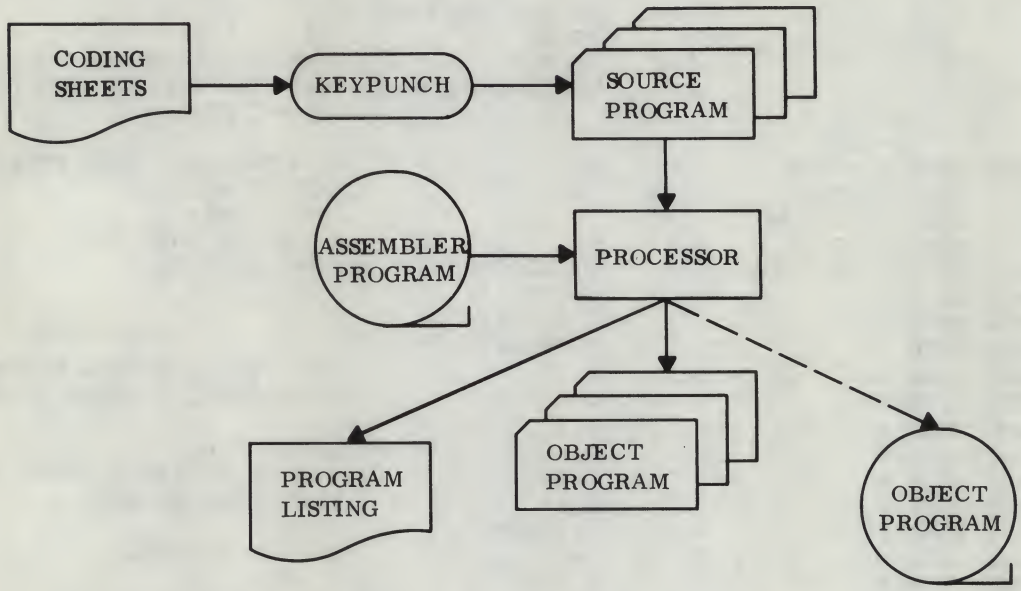


Figure 3-3. Programming Process

3.2 ASSEMBLER CODING CONVENTIONS

Assembly language programs are structured around source statements that contain from one to five fields as follows: label (optional), operation (mandatory), operand (usually required), comment (optional), and identification sequence (optional). The fields must be entered in the following order.

[label field] operation field operand field [comment field] [identification field]

The sample coding form shown in figure 3-1 has the five fields delineated; however, since the assembler program accepts "free-form" statements, the programmer is permitted to disregard field boundaries. For clarity and readability, use of field boundaries, wherever possible, is highly recommended.

The entry in each of the five fields must meet certain specifications, and, in many cases, the programmer must understand how the assembler program executes certain types of instructions in order to code legal statements. The following paragraphs describe the entry requirements for the five fields.

3.2.1 Label Field

The label field is optional and may contain a symbol used to identify the current statement when referenced by other statements. More than one label may appear in the label field, in which case any of the labels may be used to reference the labeled location. Each label in the label field is terminated by a colon (:). For example:

```
A: B: C:    LD    AC0,(AC2)
```

A label may appear by itself in a statement; in which case, it refers to the next instruction or data word in the source program. For example:

```
START:
            ST    AC0,STADD (AC2)
```

3.2.2 Operation Field

The operation field is mandatory and contains a mnemonic that defines an assembler operation (such as a directive) or a machine operation (such as a load).

Instruction statements define the machine operations. Valid instruction mnemonics are listed in appendixes B and C. Directive statements control the process of program assembly and may generate data. Valid directive mnemonics are listed in appendix D.

The operation field must be terminated by one or more blanks.

3.2.3 Operand Field

The operand field contains entries that identify data to be acted upon by the statement. A space is not required to terminate the field. An operand entry is composed of one or more terms that represent a value. The value may be inherent in the term, in which case the term is a constant (3.3.2.1), or the value may be assigned by the assembler program during assembly; in which case, the term is symbolic (3.3.2.2). An arithmetic combination of terms is reduced to a single value by the assembler program as described in 3.3.3. The relationship of terms is shown in figure 3-5.

3.2.4 Comment Field

Comments are optional descriptive notes printed on the program listing for programmer reference. Comments should be included throughout the program to explain subroutine linkages, assumptions made, formats of inputs processed, and so forth. A comment may follow a statement, or the comment may be entered on a separate statement line(s) since the comment has no affect on the assembled program and is printed only on the program listing.

The following conventions apply to comments:

1. A comment must be preceded by a semicolon (;).
2. All valid characters, including blanks, may be used in comments.
3. Comments should not extend beyond column 72, but a comment may be carried over on the following line (preceded by a semicolon).

3.2.5 Identification Sequence Field

The identification sequence field is an optional entry that specifies program identification and/or statement sequence characters. If the field or a portion of the field is used for program identification, the identification is punched in the statement cards and is listed on the program listing.* This field is generally not used with paper tape input.

As an aid to keeping source statements in order, the programmer may code a sequence of characters in ascending order in the identification sequence field.

The identification sequence field is fixed in columns 73 through 80 of the source image. Columns 73 through 80 are ignored by the assembler but are printed in the program listing.*

3.2.6 Example Statement

An example assembler statement is as follows:

Label	Operation	Operand	Comment
LOOP:	LD	AC0,1(AC2)	;GET A VALUE

The label, LOOP, is used to refer to the example statement in later (or previous) statements; in effect, to loop to the statement. The mnemonic operation code, LD, stipulates the type of operation. The operand field specifies an Accumulator (AC0), an index register (AC2), and a displacement (+1), and comment field (which contains a note that may be used by the programmer to identify quickly the action defined by the statement). See chapter 7 for other statement examples.

3.3 BASIC ELEMENTS

PACE assembly language statements have well-defined formats constructed from the following elements.

3.3.1 Character Set

Assembly language statements are written using the following letters, numbers, and special characters.

Letters:	A through Z
Numbers:	0 through 9
Special Characters:	! $ % & ′ ( ) * + , - . / : ; < = > @ ␣ # ∧
	Note: ␣ denotes a blank

Except for the lower-case letters, any of the printable characters listed in appendix A, "ASCII Character Set in Hexadecimal Representation," may be specified with a .ASCII directive statement (see 5.3.9 for a description of the .ASCII directive).

Example:

```
.ASCII   'ORDER # 32/65'
```

Nonprintable or printable characters may be specified with a .WORD directive statement. Directive statements are discussed in chapter 5.

Example:

```
.WORD    0A              ;ASCII CARRIAGE RETURN
```

* In some cases, the width of the carriage of the output device that prints the listing is not wide enough to allow inclusion of the information in the identification sequence field in the listing.

3.3.2 Terms

The relationship of terms is shown in figure 3-4. The various types of terms are described in the following paragraphs.

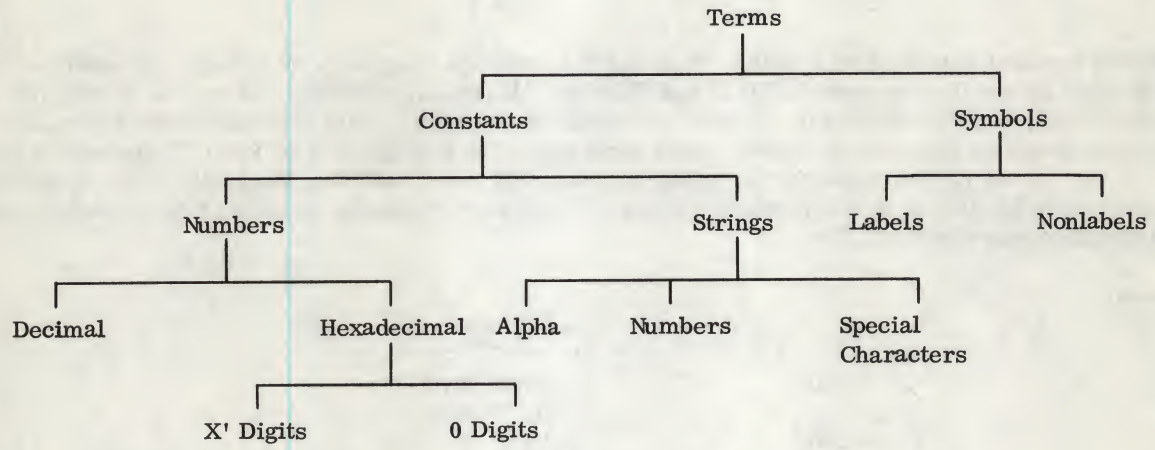


Figure 3-4. Relationship of Terms

3.3.2.1 Constants

A self-defining term, or constant, has its value inherent in the term. The assembler program does not assign a value to the term, but derives the value from the term.

Constants are used to specify immediate data, addresses, registers, and input/output information to the assembler program. Three types of constants are available: decimal, hexadecimal, and character (or string).

A decimal constant is zero or a decimal integer that does not begin with zero. For 16-bit data the value range is 0 to 65,535 for an unsigned decimal integer and +32,767 to -32,768 for a signed decimal integer. For 8-bit data, the value range is 0 to 255 for an unsigned decimal integer and +127 to -128 for a signed decimal integer. It should be noted that having signed and unsigned data are just a coding convenience made available because some instructions treat data as signed values and others treat data as signed values. For example, in 8-bit data, -1_{10} and 255_{10} both convert to FF_{16} internally. The processor itself always does signed arithmetic, except in BCD.

Examples:

```
                          decimal constants
                         /                 \
.WORD                    0, 40000, -3165, +32
LI                       0,0
```

A hexadecimal self-defining term may be specified in either of two ways. The term may start with X'; or the term may start with a leading zero. The range of hexadecimal numbers is 0 to $FFFF_{16}$ for 16-bit data and 0 to FF_{16} for 8-bit data.

Examples:

```
                                    hexadecimal constants
                                    ^^^^^^^^^^^^^^^^^^^^^
.WORD                               X'FF,X'10
.WORD                               X'1FE,X'FFFF
AISZ                                1, 01F
LI                                  0, X'40
```

A character constant is defined as a string. A string is a series of characters or a single character enclosed in single quote marks (for example, 'THIS IS A STRING'). All letters, numbers, and special characters (including blanks) may be specified in a string. If a single quote mark is part of the character string, it should immediately be preceded by another single quote mark; for example, 'DON''T DO IT' represents DON'T DO IT. A null string ('') will cause the assembler to generate a word containing two blanks. String characters are translated to ASCII code (see appendix A) in memory with each character occupying 8 bits. Refer to the .ASCII directive described in 5.3.9.

Examples:

```
                                    ASCII constants
                                    ^^^^^^^^^^^^^^^
.ASCII                              'NUMBER'
LI                                  0, '?'/256
.WORD                               'TY'
```

3.3.2.2 Symbols

A symbol is a character or a combination of characters used to identify a memory location, a register, or any other program element. Symbolic representation of elements is superior to numeric representation for the following reasons:

1. You can give meaningful names to the elements in the program.
2. You can debug a program more easily, because the symbols are referenced in the map at the end of the program.
3. You can maintain a program more easily, because you can change a symbolic value in one place and its value will be changed throughout the program.

Symbols are defined by one of two methods:

1. By appearing in a label field in a statement (see 3.2.1).

   ```
   symbol
   ^^^^^^
   SUB1:        LI        0,0              ;CLEAR ACCUMULATOR 0
   ```

   The value assigned to a symbol appearing in a label field is the address of the instruction, data, or storage location named by the symbol.

2. By using an assignment statement to assign a specific value to a symbol (see 5.2).

   ```
   symbol
   ^^^^^^
   AC2          =         2                ;ACCUMULATOR 2
   ```

A symbol that is used to reference a location or a value may be further identified as a global symbol by the .GLOBL directive (see 5.3.12), thereby permitting other programs to access the value of the symbol.

If the address of an item changes upon program relocation, the symbol is considered a relocatable term. If the address does not change upon program relocation, the symbol is an absolute term.

Symbol construction must meet the following restrictions:

1. A symbol may contain one or more alphanumeric characters, the first of which must be either a letter or a dollar sign ($).
2. Although up to 32 letters may be included, only the first 6 letters are recognized by the assembler program. Therefore, the programmer must ensure that a long symbol is unique in the first six characters.
3. If the first character in the symbol is a dollar sign ($), the symbol is defined as a local symbol. The .LOCAL directive allows the programmer to specify that local symbols appearing between two .LOCAL directive statements have a certain meaning only within that region of the program. This enables the programmer to use common mnemonics throughout a program without causing a conflict of names.

   NOTE

   A long local symbol must be unique in the first five characters.

4. No special characters or embedded blanks may appear within a symbol.
5. Symbol values cannot exceed a positive value of 65,535 or a negative value of 32,768 for 16-bit data or addresses, or 255 and 128, respectively, for 8-bit data.

Some examples of symbols follow:

Legal Symbols	Illegal Symbols	Reason Illegal
$ABC	LONGSYMBOL1	First six characters are not unique.
LONGSYMBOL	LONGSYMBOL2	
$ABC2	2AB	First character must be alphabetic or a dollar sign ($).
$2	#CDE	
XYZ	XYZ$	Last character is not alphanumeric.
$ABCDEF	$ABCDE	First five characters of the local symbols are not unique.
$ABC2EF	$ABCDF	

3.3.3 Expressions

Operand entries (see 3.2.3), consisting of either single term or an arithmetic or a logical combination of terms, are called expressions. Expressions are either simple or multiterm. Simple expressions are single terms, such as a symbol or a constant. Multiterm expressions are simple expressions that are combined using the arithmetic and logical operators shown in table 3-1. The multiterm expression is evaluated by the assembler program in a left-to-right order regardless of the operators used between the terms. Parentheses are not permitted for the purpose of grouping arithmetic and/or logical operations; they have special significance in defining certain assembler functions.

The result of the expression evaluation is a 16-bit value.

Examples:

```
TBLEND:   .WORD       TABLE + X'10
          SKNE        AC0, TBL + 3
          .SET ENTRY, ENTRY1 + ENTRY2 - 4
```

3.3.3.1 Arithmetic and Logical Operators

Table 3-1 lists the arithmetic and logical operators available for forming expressions.

Table 3-1. Arithmetic and Logical Operators

Operator	Function	Type
+	Addition	Binary
-	Subtraction	Unary or binary
*	Multiplication	Binary
/	Division	Binary
%	Logical NOT	Unary
&	Logical AND	Binary
!	Logical OR	Binary
<	Less than	Binary
=	Equal to	Binary
>	Greater than	Binary

A unary operator operates upon one operand and appears in the format 'op opnd' (for example, -9). A binary operator operates upon two operands and appears in the format '$opnd_1$ op $opnd_2$' (for example, A&B).

In the expressions "A < B", "A = B", and "A > B" the result is a one if the condition is true and a zero if the condition is false.

3.3.4 Literals

A literal is a constant, symbol or expression in an operand field that represents a value literally rather than the address of a value. For example, a literal 7 represents the value 7.

The format for literals is

```
=literal
```

The address field of any single word instruction, if preceded by an equal sign, becomes a literal value. The value of the expression is allocated memory in the same way as assembler-generated indirect pointers. For example:

```
        SKNE    AC0,=1000
```

will generate code similar to the programmer-specified

```
        SKNE    AC0,D1000
                 :
D1000:  .WORD   1000
```

Specifying =SYMBOL will cause a reference to a word containing the address of SYMBOL.

NOTE

In assembly language, BCD literals must be specified as hexadecimal. For example:

```
DECA    AC0,=099        ;ADD BCD 99 TO AC0
```

Chapter 4

INSTRUCTION SET

The assembly language instruction set of PACE provides arithmetic, logic, branch, skip, shift, and other operations between the accumulators and memory and other registers.

Instruction statements, when assembled, generate the object (machine) code that defines the operations the processor will perform.

The PACE instruction set is summarized in table 2-3 and appendix B. Refer to table 2-4 for definitions of the symbols used in the notation for describing the PACE instruction set.

4.1 BRANCH INSTRUCTIONS

The seven instructions in this group are used for conditional and unconditional jumps within a routine, jumps to subroutines, and returns from subroutines and interrupts. The Branch Instructions and mnemonics are as follows:

Branch on Condition	BOC
Jump	JMP
Jump Indirect	JMP@
Jump to Subroutine	JSR
Jump to Subroutine Indirect	JSR@
Return from Subroutine	RTS
Return from Interrupt	RTI

NOTE

JMP@ and JSR@ are specified to the assembler as JMP and JSR with indirection specified by the address field.

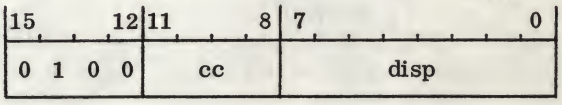
The source statement format, the instruction format and the description of the operation of each Branch Instruction follows.

BRANCH ON CONDITION (BOC)

SOURCE STATEMENT

Mnemonic	Operands
BOC	cc,address

INSTRUCTION FORMAT



Operation: If cc true, (PC) ← (PC) + disp, B

If the condition identified by the code, cc, is true, the value of the displacement, disp, is added to the contents of the Program Counter, (PC), and the address formed is stored in PC. The next instruction is fetched from the location designated by the new contents of PC. If cc is not true, PC is incremented by 1 and the instruction following the BOC is executed. The condition codes are listed in table 4-1.

NOTE

PC addresses the location following the BOC when the addition occurs (that is, the branch is relative to the next instruction after the BOC).

Address Class: 3

Table 4-1. Branch Conditions

Condition Code (cc)	Mnemonic	Condition
0000	STFL	Stack full.
0001	REQ0	(AC0) equal to zero (1).
0010	PSIGN	(AC0) has positive sign (2).
0011	BIT0	Bit 0 of AC0 set.
0100	BIT1	Bit 1 of AC0 set.
0101	NREQ0	(AC0) is nonzero (1).
0110	BIT2	Bit 2 of AC0 is set.
0111	CONTIN	CONTIN (continue) input is high.
1000	LINK	LINK is set.
1001	IEN	IEN is set.
1010	CARRY	CARRY is set.
1011	NSIGN	(AC0) has negative sign (2).
1100	OV	OV is set.
1101	JC13	JC13 input is high.
1110	JC14	JC14 input is high.
1111	JC15	JC15 input is high.

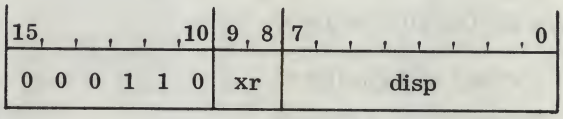
NOTES:
1. If selected data length is 8 bits, only bits 7 through 0 of AC0 are tested.
2. Bit 7 is sign bit (instead of bit 15) if selected data length is 8 bits.

JUMP (JMP)

SOURCE STATEMENT

Mnemonic	Operands
JMP	address disp(xr)

INSTRUCTION FORMAT



Operation: (PC) ← EA

The effective Address, EA, replaces the contents of the Program Counter, (PC). The next instruction is fetched from the location designated by the new contents of PC.

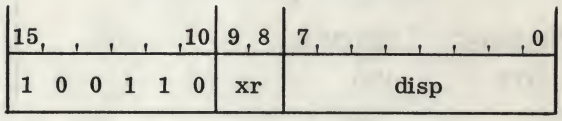
Address Class: 2

JUMP INDIRECT (JMP@)

SOURCE STATEMENT

Mnemonic	Operands
JMP	@address @disp(xr)

INSTRUCTION FORMAT



Operation: (PC) ← (EA)

The contents of the Effective Address, (EA), replace the contents of the Program Counter, (PC). The next instruction is fetched from the location designated by the new contents of PC.

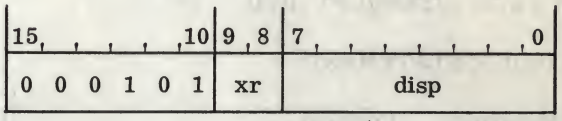
Address Class: 2

JUMP TO SUBROUTINE (JSR)

SOURCE STATEMENT

Mnemonic	Operands
JSR	address disp(xr)

INSTRUCTION FORMAT



Operation: (STK) ← (PC), (PC) ← EA

The contents of the Program Counter, (PC), are stored on the top of the Stack, (STK). The Effective Address, EA, replaces the contents of PC. The next instruction is fetched from the location designated by the new contents of PC.

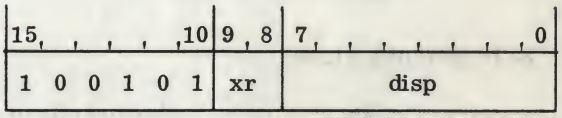
Address Class: 2

JUMP TO SUBROUTINE INDIRECT (JSR@)

SOURCE STATEMENT

Mnemonic	Operands
JSR	@address @disp(xr)

INSTRUCTION FORMAT



Operation: (STK) ← (PC), (PC) ← (EA)

The contents of the Program Counter, (PC), are stored on the top of the Stack, (STK). The contents of the Effective Address, (EA), replace the contents of PC. The next instruction is fetched from the location designated by the new contents of PC.

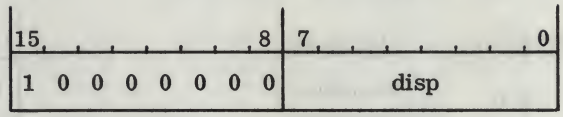
Address Class: 2

RETURN FROM SUBROUTINE (RTS)

SOURCE STATEMENT

Mnemonic	Operand
RTS	disp

INSTRUCTION FORMAT



Operation: (PC) ← (STK) + disp

The sum of the contents of the top of the Stack, (STK), and the displacement, disp, replaces the contents of the Program Counter, (PC). The next instruction is fetched from the location designated by the new contents of PC.

NOTE

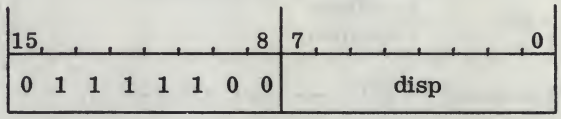
RTS is used primarily to return from subroutines entered by JSR.

RETURN FROM INTERRUPT (RTI)

SOURCE STATEMENT

Mnemonic	Operand
RTI	disp

INSTRUCTION FORMAT



Operation: (PC) ← (STK) + disp, IEN = 1

The sum of the contents of the top of the Stack, (STK), and the displacement, disp, replaces the contents of the Program Counter, (PC). The Interrupt Enable Flag, IEN, is set. The next instruction is fetched from the location designated by the new contents of PC.

NOTE

RTI is used primarily to exit from an interrupt service routine.

4.2 SKIP INSTRUCTIONS

The six instructions in this group are used primarily to control loops and maintain counters. The Skip Instructions and mnemonics are as follows:

Skip if Not Equal	SKNE
Skip if Greater	SKG
Skip if AND is Zero	SKAZ
Increment and Skip if Zero	ISZ
Decrement and Skip if Zero	DSZ
Add Immediate, Skip if Zero	AISZ

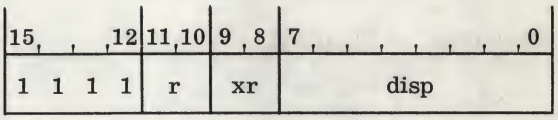
The format of the source statement, the instruction format, and the description of the operation of each Skip Instruction follows.

SKIP IF NOT EQUAL (SKNE)

SOURCE STATEMENT

Mnemonic	Operands
SKNE	r,address r,disp(xr)

INSTRUCTION FORMAT



Operation: If (ACr) $\neq$ (EA), (PC) $\leftarrow$ (PC) + 1, B

If the contents of Accumulator r, (ACr), do not equal the contents of the Effective Address, (EA), the next instruction in sequence is skipped. The contents of ACr and the contents of EA are not altered. If 8-bit data length is selected (BYTE = 1 in FR), only bits 7 through 0 are compared.

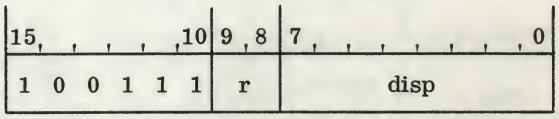
Address Class: 1

SKIP IF GREATER (SKG)

SOURCE STATEMENT

Mnemonic	Operands
SKG	0,address 0,disp(xr)

INSTRUCTION FORMAT



Operation: If (AC0) > (EA), (PC) $\leftarrow$ (PC) + 1, B

The contents of Accumulator 0, (AC0), and the contents of the Effective Address, (EA), are compared as 16-bit signed numbers. If the contents of AC0 is greater (more positive) than the contents of EA, the next instruction is skipped. The contents of AC0 and the contents of EA are not altered. If 8-bit data length is selected (BYTE = 1 in FR), only bits 7 through 0 are compared.

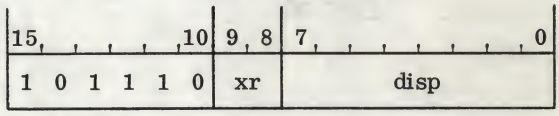
Address Class: 1

SKIP IF AND IS ZERO (SKAZ)

SOURCE STATEMENT

Mnemonic	Operands
SKAZ	0,address 0,disp(xr)

INSTRUCTION FORMAT



Operation: If [(AC0) $\wedge$ (EA)] = 0, (PC) $\leftarrow$ (PC) + 1, B

The contents of Accumulator 0, (AC0), are ANDed with the contents of the Effective Address, (EA). If the result equals zero, the next instruction in sequence is skipped. The contents of AC0 and the contents of EA are not altered. If 8-bit data length is selected (BYTE = 1 in FR), only bits 7 through 0 are tested. This instruction may be used to test one or more bits in a memory word.

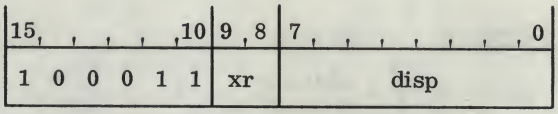
Address Class: 1

INCREMENT AND SKIP IF ZERO (ISZ)

SOURCE STATEMENT

Mnemonic	Operands
ISZ	address disp(xr)

INSTRUCTION FORMAT



Operation: (EA) ← (EA) + 1; if (EA) = 0, (PC) ← (PC) + 1, B

The contents of the Effective Address, (EA), are incremented by 1. If the new contents of EA equal zero, the next instruction in sequence is skipped. If 8-bit data length is selected (BYTE = 1 in FR), only bits 7 through 0 are tested.

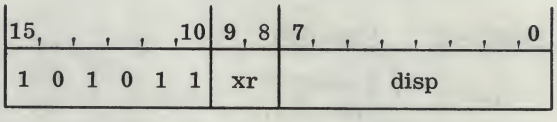
Address Class: 1

DECREMENT AND SKIP IF ZERO (DSZ)

SOURCE STATEMENT

Mnemonic	Operands
DSZ	address disp(xr)

INSTRUCTION FORMAT



Operation: (EA) ← (EA) - 1; if (EA) = 0, (PC) ← (PC) + 1, B

The contents of the Effective Address, (EA), are decremented by 1. If the new contents of EA equal zero, the next instruction in sequence is skipped. If 8-bit data length is selected (BYTE = 1 in FR), only bits 7 through 0 are tested.

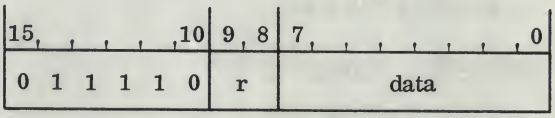
Address Class: 1

ADD IMMEDIATE, SKIP IF ZERO (AISZ)

SOURCE STATEMENT

Mnemonic	Operands
AISZ	r, data

INSTRUCTION FORMAT



Operation: (ACr) ← (ACr) + data; If (ACr) = 0, (PC) ← (PC) + 1

The contents of Accumulator r, (ACr), are replaced by the sum of the contents of ACr and data. The initial contents of ACr are lost. If the new contents of ACr equal zero, the contents of the Program Counter (PC) are incremented by 1, skipping the next instruction in sequence. This instruction always tests the full 16-bit result, independent of the data length selected.

NOTE

Testing the 16-bit result in conjunction with no change to the status indicators allows AISZ to be conveniently used for modifying 16-bit index values while working with 8-bit data.

4.3 MEMORY DATA-TRANSFER INSTRUCTIONS

The five instructions in this group are used for data transfers between registers and memory or peripherals. The Memory Data-Transfer Instructions and mnemonics are as follows:

Load	LD
Load Indirect	LD@
Store	ST
Store Indirect	ST@
Load with Sign Extended	LSEX

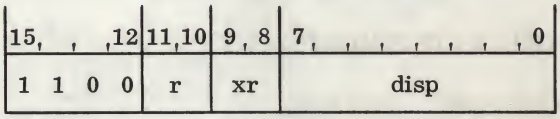
The source statement format, instruction format and the description of each Memory Data-Transfer Instruction follows.

LOAD (LD)

SOURCE STATEMENT

Mnemonic	Operands
LD	r,address r,disp(xr)

INSTRUCTION FORMAT



Operation: (ACr) ← (EA); EA = (xr) + disp

The contents of the Effective Address, (EA), replace the contents of Accumulator r, (ACr). EA is the address formed by the sum of the contents of the index register, (xr), and the displacement, disp. The initial contents of ACr are lost; the contents of EA are not altered.

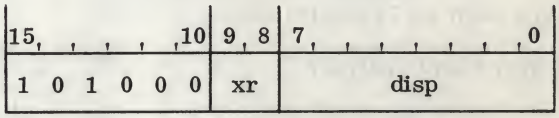
Address Class: 2

LOAD INDIRECT (LD@)

SOURCE STATEMENT

Mnemonic	Operands
LD	0,@address 0,@disp(xr)

INSTRUCTION FORMAT



Operation: (AC0) ← (EA); EA = ((xr) + disp)

The contents of the Effective Address, (EA), replace the contents of Accumulator 0, (AC0). EA is the contents of the address formed by the sum of the contents of the index register, (xr), and the displacement, disp. The initial contents of AC0 are lost; the contents of EA and the location that designates EA are not altered.

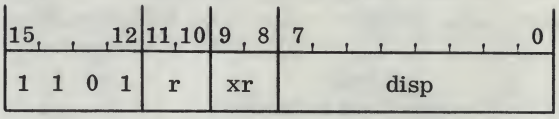
Address Class: 2

STORE (ST)

SOURCE STATEMENT

Mnemonic	Operands
ST	r,address r,disp(xr)

INSTRUCTION FORMAT



Operation: (EA) ← (ACr); EA = (xr) + disp

The contents of Accumulator r, (ACr), replace the contents of the Effective Address, (EA). EA is the address formed by the sum of the contents of the index register, (xr), and the displacement, disp. The initial contents of EA are lost; the contents of ACr are not altered.

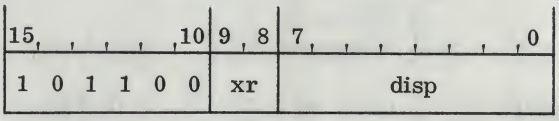
Address Class: 2

STORE INDIRECT (ST@)

SOURCE STATEMENT

Mnemonic	Operands
ST	0,@address 0,@disp(xr)

INSTRUCTION FORMAT



Operation: (EA) ← (AC0); EA = ((xr) + disp)

The contents of Accumulator 0, (AC0), replace the contents of the Effective Address, (EA). EA is the contents of the address formed by the sum of the contents of the index register, (xr), and the displacement, disp. The initial contents of EA are lost; the contents of AC0 and the location that designates EA are not altered.

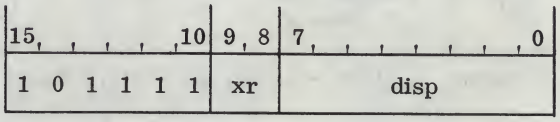
Address Class: 2

LOAD WITH SIGN EXTENDED (LSEX)

SOURCE STATEMENT

Mnemonic	Operands
LSEX	0,address 0,disp(xr)

INSTRUCTION FORMAT



Operation: (AC0) ← (EA) bit 7 extended

The contents of Accumulator 0, (AC0), are replaced by the contents of the Effective Address, (EA), with bit 7 (the sign bit) extended through bits 15 through 8. The initial contents of AC0 are lost; the contents of EA are not altered.

NOTE

The LSEX Instruction allows 8-bit arithmetic data to be loaded from an 8-bit data memory or peripheral device register and to be operated on as 16-bit arithmetic data.

Address Class: 1

4.4 MEMORY DATA-OPERATE INSTRUCTIONS

The five instructions in this group provide arithmetic and logic operations between registers and memory. The Memory Data-Operate Instructions and mnemonics are as follows:

AND	AND
OR	OR
Add	ADD
Subtract with Borrow	SUBB
Decimal Add	DECA

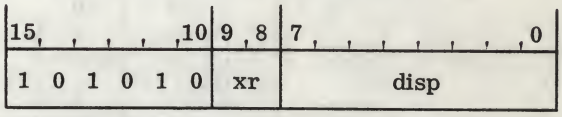
The source statement format, the instruction format and the description of the operation of each Memory Data-Operate Instruction follows.

AND (AND)

SOURCE STATEMENT

Mnemonic	Operands
AND	0,address 0,disp(xr)

INSTRUCTION FORMAT



Operation: (AC0) ← (AC0) ∧ (EA)

The contents of Accumulator 0, (AC0), and the contents of the Effective Address, (EA), are ANDed, and the result is stored in AC0. The initial contents of AC0 are lost; the contents of EA are not altered. This instruction is generally used to clear data bits. The truth table for this instruction is shown below.

$AC0_b$	EA_b	$AC0_b \wedge EA_b$
0	0	0
0	1	0
1	0	0
1	1	1

b = bits 15 to 0

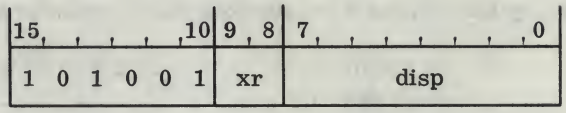
Address Class: 1

OR (OR)

SOURCE STATEMENT

Mnemonic	Operands
OR	0,address 0,disp(xr)

INSTRUCTION FORMAT



Operation: (AC0) ← (AC0) ∨ (EA)

The contents of Accumulator 0, (AC0), and the contents of the Effective Address, (EA), are inclusively-ORed. The result is stored in AC0. The initial contents of AC0 are lost; the contents of EA are not altered. This instruction is generally used to set data bits. The truth table for this instruction is shown below.

$AC0_b$	EA_b	$AC0_b \vee EA_b$
0	0	0
0	1	1
1	0	1
1	1	1

b = bits 15 to 0

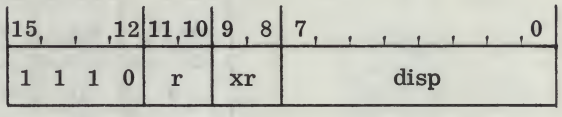
Address Class: 1

ADD (ADD)

SOURCE STATEMENT

Mnemonic	Operands
ADD	r,address r,disp(xr)

INSTRUCTION FORMAT



Operation: (ACr) ← (ACr) + (EA); CY, OV

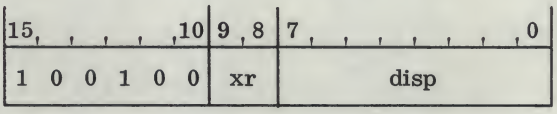
The contents of Accumulator r, (ACr), are added to the contents of the Effective Address, (EA), and the sum is stored in ACr. The Carry Flag, CY, is set if a carry from the most significant bit position occurs; otherwise, it is cleared. The Overflow Flag, OV, is set if an overflow occurs; otherwise it is cleared (see table 2-2 for definition of overflow).

SUBTRACT WITH BORROW (SUBB)

SOURCE STATEMENT

Mnemonic	Operands
SUBB	0,address 0,disp(xr)

INSTRUCTION FORMAT



Operation: (AC0) ← (AC0) + ~ (EA) + CY; CY, OV

The contents of Accumulator 0, (AC0), and the ones complement of the contents of the Effective Address, (EA), and the Carry Flag, CY, are added, and the sum is stored in AC0. The initial contents of AC0 are lost; the contents of EA are not altered. The Carry Flag is set if a carry from the most significant bit position occurs; otherwise, it is cleared. The Overflow Flag, OV, is set if an overflow occurs; otherwise, it is cleared. (See table 2-2 for definition of overflow.)

DECIMAL ADD (DECA)

SOURCE STATEMENT

Mnemonic	Operands
DECA	0,address 0,disp(xr)

INSTRUCTION FORMAT

15 ... 10	9, 8	7 ... 0
1 0 0 0 1 0	xr	disp

Operation: (AC0) ← $(AC0)_{10} + (EA)_{10}$ + CY; CY, OV

The contents of Accumulator 0, (AC0), and the contents of the Effective Address, (EA), are treated as 4-digit Binary-Coded-Decimal (BCD) numbers greater than or equal to 0, and less than or equal to 9999 ($0 \geq BCD \geq 9999$). The contents of AC0, the contents of EA, and the Carry Flag, CY, are added and the 4-digit BCD sum is stored in AC0. The initial contents of AC0 are lost; the contents of EA are not altered. CY is set if a carry occurs from the most significant decimal digit; otherwise, it is cleared. The Overflow Flag, OV, is set to an arbitrary state.

Address Class: 1

4.5 REGISTER DATA-TRANSFER INSTRUCTIONS

The ten instructions in this group are used for data transfers between registers, the stack, and the status flags register. The Register Data-Transfer Instructions and mnemonics are as follows:

Load Immediate	LI
Register Copy	RCPY
Register Exchange	RXCH
Exchange Register and Stack	XCHRS
Copy Flags into Register	CFR
Copy Registers into Flags	CRF
Push Register onto Stack	PUSH
Pull Stack into Register	PULL
Push Flags onto Stack	PUSHF
Pull Stack into Flags	PULLF

The source statement format, the instruction format and the description of the operation of each Register Data-Transfer Instruction follows.

LOAD IMMEDIATE (LI)

SOURCE STATEMENT

Mnemonic	Operands
LI	r,data

INSTRUCTION FORMAT

15 ... 10	9, 8	7 ... 0
0 1 0 1 0 0	r	data

Operation: (ACr) ← data (bit 7 extended)

The contents of Accumulator r, (ACr), are replaced by data with sign bit 7 extended through bit 15. The initial contents of ACr are lost.

REGISTER COPY (RCPY)

SOURCE STATEMENT

Mnemonic	Operands
RCPY	sr,dr

INSTRUCTION FORMAT

15 ... 10	9, 8	7, 6	5 ... 0
0 1 0 1 1 1	dr	sr	0 0 0 0 0 0

Operation: (ACdr) ← (ACsr)

The contents of the Source Register, (ACsr), replace the contents of the Destination Register, (ACdr). The initial contents of ACdr are lost; the contents of ACsr are not altered.

REGISTER EXCHANGE (RXCH)

SOURCE STATEMENT

Mnemonic	Operands
RXCH	sr,dr

INSTRUCTION FORMAT

15 ... 10	9, 8	7, 6	5 ... 0
0 1 1 0 1 1	dr	sr	0 0 0 0 0 0

Operation: (ACsr) ↔ (ACdr)

The contents of the Source Register, (ACsr), and the contents of the Destination Register, (ACdr), are exchanged.

EXCHANGE REGISTER AND STACK (XCHRS)

SOURCE STATEMENT

Mnemonic	Operand
XCHRS	r

INSTRUCTION FORMAT

15 ... 10	9, 8	7 ... 0
0 0 0 1 1 1	r	0 0 0 0 0 0 0 0

Operation: (STK) ↔ (ACr)

The contents of the top of the Stack, (STK), and the contents of Accumulator r, (ACr), are exchanged.

NOTE

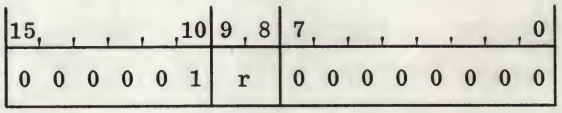
The XCHRS Instruction provides a convenient means of placing a subroutine return address into an index register for modification and/or use in passing parameters.

COPY FLAGS TO REGISTER (CFR)

SOURCE STATEMENT

Mnemonic	Operand
CFR	r

INSTRUCTION FORMAT



Operation: (ACr) ← (FR)

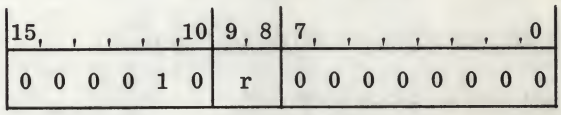
The contents of the Status Flags Register, (FR), replace the contents of Accumulator r, (ACr). The initial contents of ACr are lost; the contents of FR are not altered. The FR is described in 2.3.2 and table 2-2.

COPY REGISTER TO FLAGS (CRF)

SOURCE STATEMENT

Mnemonic	Operand
CRF	r

INSTRUCTION FORMAT



Operation: (FR) ← (ACr)

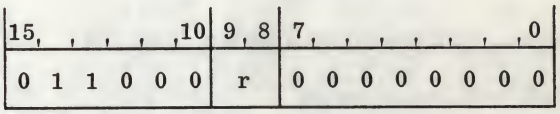
The contents of Accumulator r, (ACr), replace the contents of the Status Flags Register, (FR). The initial contents of FR are lost; the contents of ACr are not altered. The FR is described in 2.3.2 and table 2-2.

PUSH REGISTER ONTO STACK (PUSH)

SOURCE STATEMENT

Mnemonic	Operand
PUSH	r

INSTRUCTION FORMAT



Operation: (STK) ← (ACr)

The contents of Accumulator r, (ACr), are pushed onto the top of the Stack, (STK), and the stack pointer is incremented by 1. The contents of ACr are not altered.

NOTE

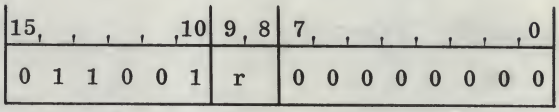
If PUSH causes the internal stack pointer to go to X'8 (nine words on the stack), the Stack-full Interrupt Request is set and the Stack-full Jump Condition is set.

PULL STACK INTO REGISTER (PULL)

SOURCE STATEMENT

Mnemonic	Operand
PULL	r

INSTRUCTION FORMAT



Operation: (ACr) ← (STK)

The contents from the top of the Stack, (STK), replace the contents of Accumulator r, (ACr). The initial contents of ACr are lost. The contents of the stack pointer are decremented by 1.

NOTE

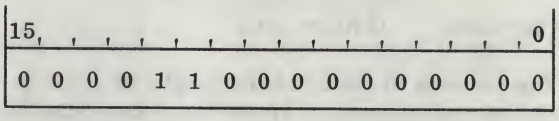
If the internal stack pointer goes to -1 (that is, no words left on stack) a Stack-empty Interrupt Request is generated.

PUSH FLAG REGISTER ONTO STACK (PUSHF)

SOURCE STATEMENT

Mnemonic
PUSHF

INSTRUCTION FORMAT



Operation: (STK) ← (FR)

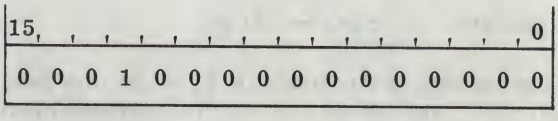
The contents of the Status Flags Register, (FR), are pushed onto the top of the Stack, (STK), and the stack pointer is incremented by 1. The contents of FR are not altered. The FR is described in 2.3.2 and table 2-2.

PULL FLAG REGISTER FROM STACK (PULLF)

SOURCE STATEMENT

Mnemonic
PULLF

INSTRUCTION FORMAT



Operation: (FR) ← (STK)

The contents of the top of the Stack, (STK), replaces the contents of the Status Flags Register, (FR). The initial contents of FR are lost. The Stack Pointer is decremented by 1. The FR is described in 2.3.2 and table 2-2.

4.6 REGISTER DATA-OPERATE INSTRUCTIONS

The five instructions in this group provide for register-to-register arithmetic and logic operations. The Register Data-Operate Instructions are as follows:

Register Add	RADD
Register Add with Carry	RADC
Register AND	RAND
Register Exclusive-OR	RXOR
Complement and Add Immediate	CAI

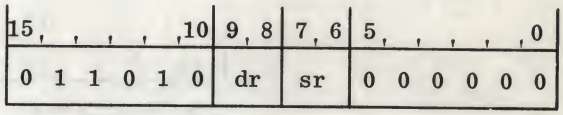
The source statement format, the instruction format and the description of the operation of each Register Data-Operate Instruction follows.

REGISTER ADD (RADD)

SOURCE STATEMENT

Mnemonic	Operands
RADD	sr,dr

INSTRUCTION FORMAT



Operation: (ACdr) ← (ACdr) + (ACsr); CY, OV

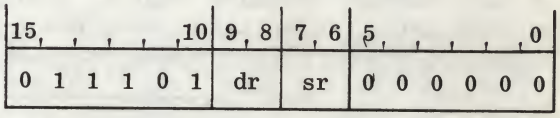
The contents of the Destination Register, (ACdr), are added to the contents of the Source Register, (ACsr), and the sum is stored in ACdr. The initial contents of ACdr are lost; the contents of ACsr are not altered. The Carry Flag, CY, is set if a carry from the most significant bit position occurs; otherwise, it is cleared. The Overflow Flag, OV, is set if an overflow occurs; otherwise, it is cleared. (See table 2-2 for definition of overflow.)

REGISTER ADD WITH CARRY (RADC)

SOURCE STATEMENT

Mnemonic	Operands
RADC	sr,dr

INSTRUCTION FORMAT



Operation: (ACdr) ← (ACdr) + (ACsr) + CY; CY, OV

The contents of the Destination Register, (ACdr), the contents of the Source Register, (ACsr), and the Carry Flag, CY, are added, and the sum is stored in ACdr. The initial contents of ACdr are lost; the contents of ACsr are not altered. The Carry Flag is set if a carry from the most significant bit position occurs; otherwise, it is cleared. The Overflow Flag, OV, is set if an overflow occurs; otherwise, it is cleared. (See table 2-2 for a definition of overflow.)

REGISTER AND (RAND)

SOURCE STATEMENT

Mnemonic	Operands
RAND	sr, dr

INSTRUCTION FORMAT

15 10	9, 8	7, 6	5 0
0 1 0 1 0 1	dr	sr	0 0 0 0 0 0

Operation: (ACdr) ← (ACdr) ∧ (ACsr)

The contents of the Destination Register, (ACdr), are ANDed with the contents of the Source Register, (ACsr), and the result is stored in ACdr. The initial contents of ACdr are lost; the contents of ACsr are not altered. The truth table for this instruction is shown below.

$ACdr_b$	$ACsr_b$	$ACdr_b \wedge ACsr_b$
0	0	0
0	1	0
1	0	0
1	1	1

b = bits 15 to 0

REGISTER EXCLUSIVE-OR (RXOR)

SOURCE STATEMENT

Mnemonic	Operands
RXOR	sr, dr

INSTRUCTION FORMAT

15 10	9, 8	7, 6	5 0
0 1 0 1 1 0	dr	sr	0 0 0 0 0 0

Operation: (ACdr) ← (ACdr) ▽ (ACsr)

The contents of the Destination Register, (ACdr), are exclusively-ORed with the contents of the Source Register, (ACsr), and the result is stored in ACdr. The initial contents of ACdr are lost; the contents of ACsr are not altered. The truth table for this instruction is shown below.

$ACdr_b$	$ACsr_b$	$ACdr_b \triangledown ACsr_b$
0	0	0
0	1	1
1	0	1
1	1	0

b = bits 15 to 0

COMPLEMENT AND ADD IMMEDIATE (CAI)

SOURCE STATEMENT

Mnemonic	Operands
CAI	r, data

INSTRUCTION FORMAT

15 ... 10	9, 8	7 ... 0
0 1 1 1 0 0	r	data

Operation: (ACr) ← ~ (ACr) + data (sign extended)

The contents of Accumulator r, (ACr), are replaced by the sum of the ones complement of ACr and data (sign bit 7 extended through bit 15). The initial contents of ACr are lost.

NOTE

Values of 0 and 1 in the data field produce the ones and and twos complement, respectively, of (ACr).

4.7 SHIFT AND ROTATE INSTRUCTIONS

The four instructions in this group shift and rotate registers. The Shift and Rotate Instructions and mnemonics are as follows:

Shift Left	SHL
Shift Right	SHR
Rotate Left	ROL
Rotate Right	ROR

The source statement format, the instruction format and the description of the operation of each Shift and Rotate Instruction follows.

SHIFT LEFT (SHL)

SOURCE STATEMENT

Mnemonic	Operands
SHL	r, n, ℓ

INSTRUCTION FORMAT

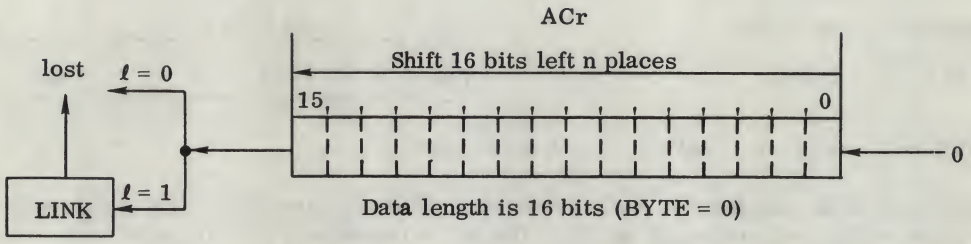
15 ... 10	9, 8	7 ... 1	0
0 0 1 0 1 0	r	n	ℓ

Operation:

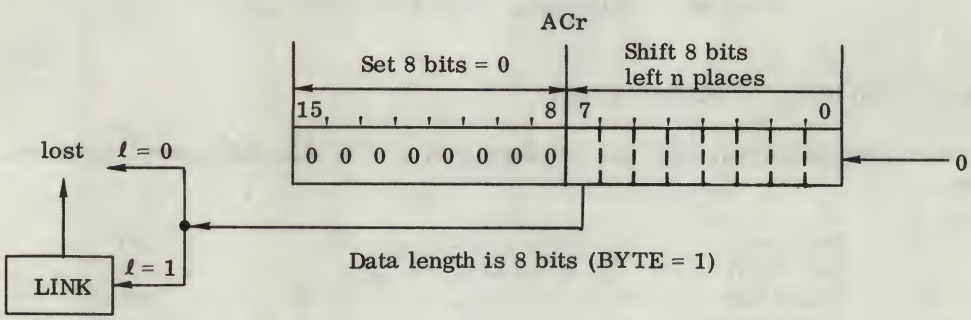
	ℓ = 0	ℓ = 1
BYTE = 0 i = 15:1	$[(ACr_i) \leftarrow (ACr_{i-1}), (ACr_0) \leftarrow 0]^n$	$[L \leftarrow (ACr_{15}), (ACr_i) \leftarrow (ACr_{i-1}), (ACr_0) \leftarrow 0]^n$
BYTE = 1 i = 7:1	$[(ACr_i) \leftarrow (ACr_{i-1}), (ACr_0) \leftarrow 0]^n$ $(ACr_{15:8}) \leftarrow 0$	$[L \leftarrow (ACr_7), (ACr_i) \leftarrow (ACr_{i-1}), (ACr_0) \leftarrow 0]^n$ $(ACr_{15:8}) \leftarrow 0$

The contents of Accumulator r, (ACr), are shifted left n bit positions; where n is 0 through 127. Zeros are shifted into the least significant bit position (ACr_0). If ℓ is zero, the most significant bit is shifted out and lost. If ℓ is one, the most significant bit is shifted into LINK, L, and the content of LINK is shifted out and lost.

If BYTE is zero, bit 15 is the most significant bit, data length is 16 bits, and 16 bits are shifted, as shown below.



If BYTE is one, bit 7 is the most significant bit and data length is 8 bits. Bits 15 through 8 are cleared and bits 7 through 0 are shifted, as shown below.

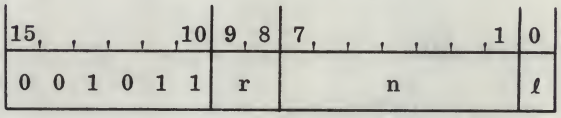


SHIFT RIGHT (SHR)

SOURCE STATEMENT

Mnemonic	Operands
SHR	r,n,ℓ

INSTRUCTION FORMAT

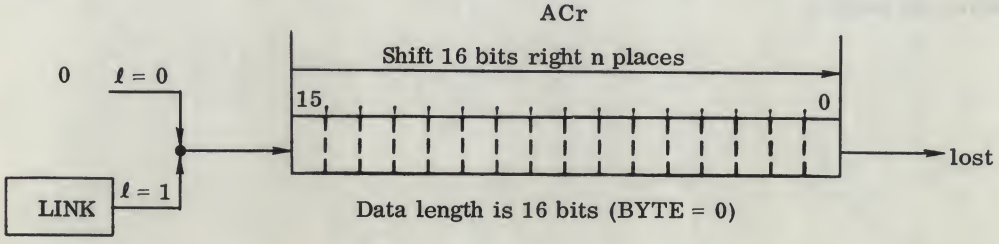


Operation:

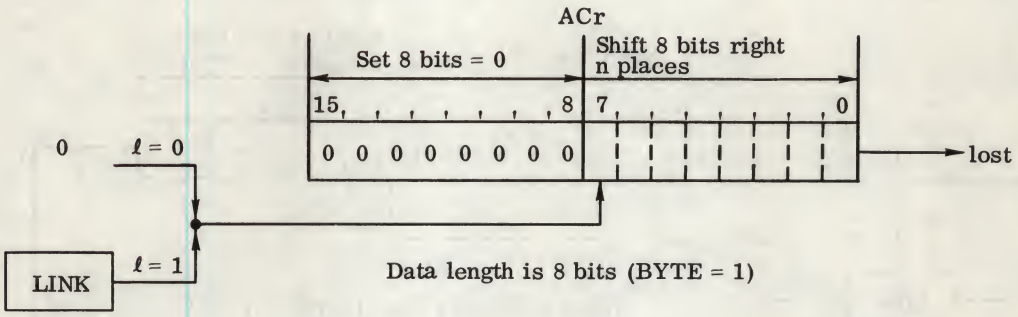
	ℓ = 0	ℓ = 1
BYTE = 0 i = 15:1	$[0 \rightarrow (ACr_{15}), (ACr_i) \rightarrow (ACr_{i-1})]^n$	$[L \rightarrow (ACr_{15}), (ACr_i) \rightarrow (ACr_{i-1})]^n$
BYTE = 1 i = 7:1	$[0 \rightarrow (ACr_7), (ACr_i) \rightarrow (ACr_{i-1})]^n$, $(ACr_{15:8}) \leftarrow 0$	$[L \rightarrow (ACr_7), (ACr_i) \rightarrow (ACr_{i-1})]^n$, $(ACr_{15:8}) \leftarrow 0$

The contents of Accumulator r, (ACr), are shifted right n bit positions; where n is 0 through 127. The bits shifted out of the least significant bit position (ACr_0) are lost. If ℓ is zero, a zero is shifted into the most significant bit position (msb). If ℓ is one, the LINK bit, L, is shifted into the most significant bit position.

If BYTE is zero, bit 15 is the most significant bit, data length is 16 bits, and 16 bits are shifted, as shown below.



If BYTE is one, bit 7 is the most significant bit and data length is 8 bits. Bits 15 through 8 are cleared and bits 7 through 0 are shifted, as shown below.

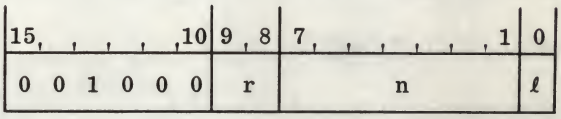


ROTATE LEFT (ROL)

SOURCE STATEMENT

Mnemonic	Operands
ROL	r,n,ℓ

INSTRUCTION FORMAT

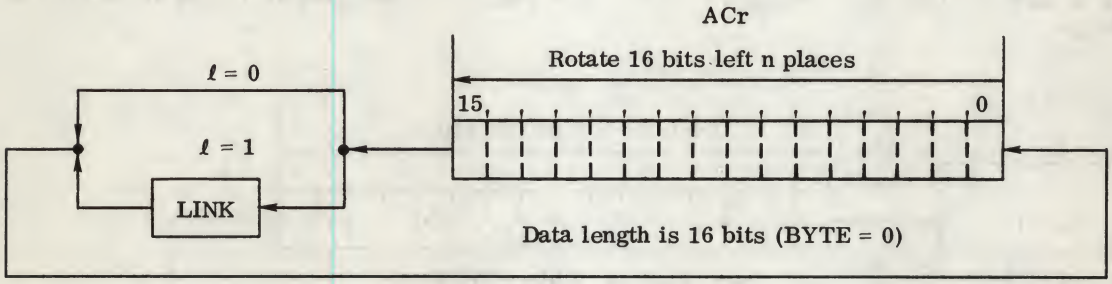


Operation:

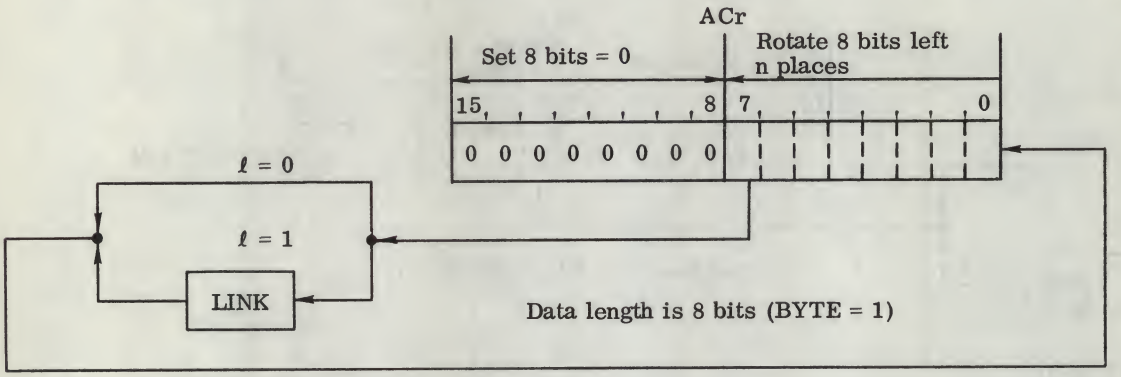
	ℓ = 0	ℓ = 1
BYTE = 0 i = 15:1	$[(ACr_i) \leftarrow (ACr_{i-1}), (ACr_0) \leftarrow (ACr_{15})]^n$	$[(ACr_i) \leftarrow (ACr_{i-1}), (ACr_0) \leftarrow L \leftarrow (ACr_{15})]^n$
BYTE = 1 i = 7:1	$[(ACr_i) \leftarrow (ACr_{i-1}), (ACr_0) \leftarrow (ACr_7)]^n$, $(ACr_{15:8}) \leftarrow 0$	$[(ACr_i) \leftarrow (ACr_{i-1}), (ACr_0) \leftarrow L \leftarrow (ACr_7)]^n$, $(ACr_{15:8}) \leftarrow 0$

The contents of Accumulator r, (ACr), are rotated left n bit positions; where n is 0 through 127. If ℓ is zero, the most significant bit is shifted into bit position 0. If ℓ is one, the most significant bit is shifted into LINK, L, and LINK is shifted into bit position 0.

If BYTE is zero, bit 15 is the most significant bit, data length is 16 bits, and 16 bits are rotated, as shown below.



If BYTE is one, bit 7 is the most significant bit and data length is 8 bits. Bits 15 through 8 are cleared and bits 7 through 0 are rotated as shown below.

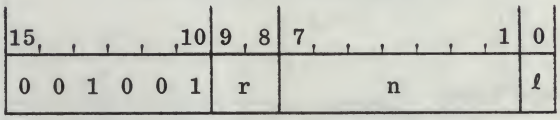


ROTATE RIGHT (ROR)

SOURCE STATEMENT

Mnemonic	Operands
ROR	r,n, ℓ

INSTRUCTION FORMAT

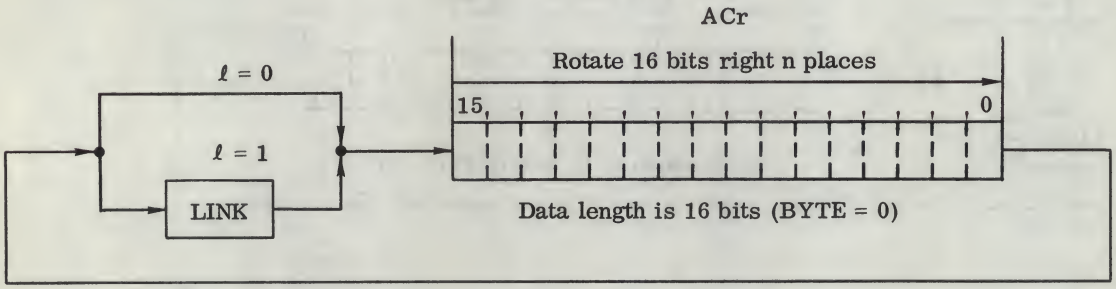


Operation:

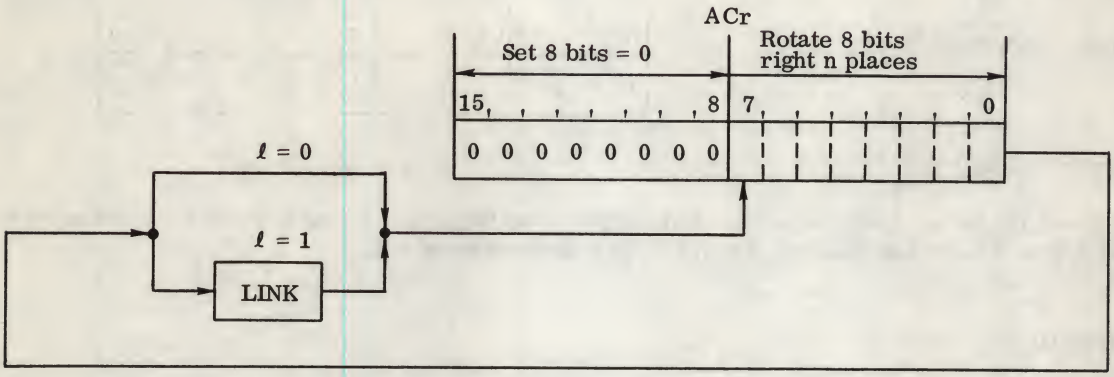
	ℓ = 0	ℓ = 1
BYTE = 0 i = 15:1	$[(ACr_i) \rightarrow (ACr_{i-1}), (ACr_0) \rightarrow (ACr_{15})]^n$	$[(ACr_i) \rightarrow (ACr_{i-1}), (ACr_0) \rightarrow L \rightarrow (ACr_{15})]^n$
BYTE = 1 i = 7:1	$[(ACr_i) \rightarrow (ACr_{i-1}), (ACr_0) \rightarrow (ACr_7)]^n$, $(ACr_{15:8}) \leftarrow 0$	$[(ACr_i) \rightarrow (ACr_{i-1}), (ACr_0) \rightarrow L \rightarrow (ACr_7)]^n$, $(ACr_{15:8}) \leftarrow 0$

The contents of Accumulator r, (ACr), are rotated right n bit positions; where n is 0 through 127. If ℓ is zero, bit 0 is shifted into the most significant bit position. If ℓ is one, bit 0 is shifted into LINK, L, and LINK is shifted into the most significant bit position.

If BYTE is zero, bit 15 is the most significant bit, data length is 16 bits, and all 16 bits are rotated as shown below.



If BYTE is one, bit 7 is the most significant bit and data length is 8 bits. Bits 15 through 8 are cleared and bits 7 through 0 are rotated, as shown below.



4.8 MISCELLANEOUS INSTRUCTIONS

There are four instructions in this group. The Miscellaneous Instructions and mnemonics are as follows:

Halt	HALT
Set Flag	SFLG
Pulse Flag	PFLG
No Operation	NOP

The source statement format, the instruction format and the description of the operation of each Miscellaneous Instruction follows.

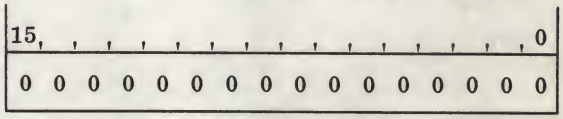
HALT (HALT)

SOURCE STATEMENT

Mnemonic

HALT

INSTRUCTION FORMAT



Operation: Halt

During normal program execution, the NHALT control line provides a high output. If a Halt Instruction is executed, the microprocessor NHALT Output is driven low to indicate that microprocessor activity is suspended until the CONTIN Input is pulsed. While PACE operation is suspended, the NHALT Output Line has a 7/8 duty cycle; that is, every eighth clock phase, the NHALT Output goes high. The NHALT 7/8 duty cycle must be accounted for if the output is used as a logic signal but is of little concern if the output drives only a halt indicator. The NHALT Output goes high after the Halt Instruction is terminated by pulsing the CONTIN Input. The CONTIN Input must go high for twelve clock cycles, minimum, and low for four clock cycles for PACE operation to resume.

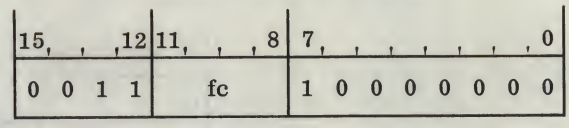
For a description of the hardware signals, see the PACE System Design Manual.

SET FLAG (SFLG)

SOURCE STATEMENT

Mnemonic	Operand
SFLG	fc

INSTRUCTION FORMAT



Operation: $(FR_{fc}) \leftarrow 1$

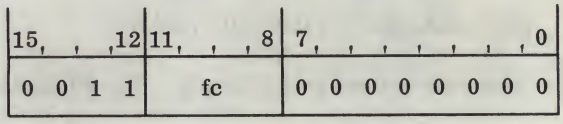
A single bit of the Status Flags Register, (FR), is set. The flag code, fc, specifies the bit that is set. All other bits of FR are not altered. See 2.3.2 for a description of FR.

PULSE FLAG (PFLG)

SOURCE STATEMENT

Mnemonic	Operand
PFLG	fc

INSTRUCTION FORMAT



Operation: $(FR_{fc}) \leftarrow 1$, $(FR_{fc}) \leftarrow 0$

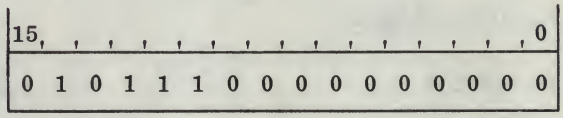
A single bit of Status Flags Register, (FR), is set, then cleared after four clock cycles. The flag code, fc, specifies which bit is pulsed. All other bits of FR are not altered. See 2.3.2 for a description of FR.

NO OPERATION (NOP)

SOURCE STATEMENT

Mnemonic
NOP

INSTRUCTION FORMAT



Operation: $(PC) \leftarrow (PC) + 1$

The contents of the Program Counter (PC) are incremented by 1.

Chapter 5

ASSEMBLER DEPENDENT STATEMENTS

Assembler-dependent statements are statements that direct the assembler to perform some particular operation. They may or may not generate any machine code. There are four types of assembler-dependent statements: the comment, the assignment, the directives, and the macros. All statements except the macros are discussed in this chapter. Because macros combine instructions and assembler-dependent statements they are discussed separately in chapter 6.

5.1 COMMENT STATEMENT

Comment statements are defined by a semicolon (;) in the first nonblank character position of the record. Comment statements do not generate code, but serve only to document the program listing output by the assembler. For example:

```
;
;THESE ARE COMMENT STATEMENTS
;
```

5.2 ASSIGNMENT STATEMENT

```
[label]        symbol = expression                    [;comments]
```

The Assignment Statement assigns the value of the expression on the right of the equals sign to the symbol on the left of the equals sign. The statement may be preceded by a series of labels.

Example:

```
RETURN = OD                        ;CARRIAGE RETURN
```

The Assignment Statement may also set or refer to the location counter in an expression. The location counter assigns relative addresses to program statements at assembly time. It is similar to the Program Counter, which contains the address of the next instruction to be executed at execution time. As each instruction or data area is assembled, the location counter is incremented by the length of the assembled item. Thus, the location counter always points to the next available location in memory.

The period '.' is a special symbol used to specify the location counter. The location-counter symbol may appear on either side of the equals sign. If it appears on the left, it is assigned the value on the right side of the equals sign. The programmer may refer to the current setting of the location counter by referencing the '.' in the expression to the right of the equals sign. Assignment statements using the location counter symbol are coded as free-form statements. For example:

```
                .=20               ;SET LOCATION COUNTER TO 20
                                   ;LOCATION CTR IN ABSOLUTE MODE
TABLE:          .=.+10             ;RESERVE 10 LOCATIONS FOR TABLE
```

If the '.' appears on the left, the expression on the right must be defined during the first pass of the assembler so subsequent label assignments may be made.

If the symbol on the left is not '.', then the expression on the right need not have a value during the first pass of the assembler, but the expression must have a value during the second pass. This permits only one level of forward referencing. An example of more than one level of forward referencing follows:

```
FST:    A=B+2    This expression undefined during pass 2.
SND:    B=C-1    This expression undefined during pass 1.
THD:    C=25     This expression absolute.
```

Labels on assignment statements are assigned the value of the location counter at the time the assignment statement is encountered.

5.3 DIRECTIVE STATEMENTS

The Directive Statements control the assembly process and may generate data in the object program. The directive operator may be preceded by one or more labels, and may be followed by a comment. It occupies the operation field and is followed by no operand or as many operands as required by the particular operator.

Assembler directive operators and their functions are summarized in table 5-1. Note that all directive operators begin with a period for easy visual differentation from the instruction operator mnemonics in the program listing. Each directive operator is described in more detail in the following paragraphs.

Table 5-1. Summary of Assembler Directives

Directive Name	Function	Page
.TITLE	Identification of program.	5-3
.SPLIT	Specifies a split base page.	5-3
.END	Physical End of source program.	5-3
.ASECT	Specifies start of an absolute section.	5-4
.BSECT	Specifies start of a base-sector-relocatable section.	5-4
.TSECT	Specifies start of a top-sector-relocatable section.	5-4
.LIST	Listing output control.	5-5
.SPACE	Space n lines in output listing.	5-6
.PAGE	Output listing to top-of-form.	5-6
.WORD	16-bit (or 8-bit) word data generation.	5-6
.ASCII	Data generation for character strings.	5-6
.SET	Assigns values to variables.	5-7
.IF	Conditional assembly.	5-7
.ELSE	Conditional assembly.	5-7
.ENDIF	Conditional assembly.	5-7
.GLOBL	Identifies global symbols.	5-8
.LOCAL	Establishes new local symbol region.	5-8
.DLEN	Specifies 8-bit or 16-bit data region.	5-9
.PTR	Generates a pointer or literal.	5-9
.POOL	Allocates pool memory.	5-9
.NOBAS	Flags assembler-generated base page pointers.	5-9
.MACRO	Begins a Macro definition.	6-2
.ENDM	Ends a macro definition.	6-2
.MLOC	Defines a macro local symbol.	6-6
.DO	Begins a macro-time do-loop.	6-7
.ENDDO	Ends a macro-time do-loop.	6-7
.EXIT	Exits a macro-time do-loop.	6-8
.IFC	Conditional assembly.	6-6
.ERROR	Macro error.	6-7
.MDEL	Delete a macro.	6-7

5.3.1 Title Directive (.TITLE)

```
[label]        .TITLE    symbol[,string]                    [;comments]
```

The .TITLE directive identifies the load module in which it appears with a symbolic name and an optional definitive title. If a .TITLE directive does not appear in the program, the load module is given the name MAINPR. If the load module contains more than one .TITLE directive, the assembler uses the last one encountered.

The symbolic name and the string must meet the symbol and string construction restrictions discussed in chapter 3.

Example:

```
.TITLE PACE, 'SAMPLE PROGRAM'
```

5.3.2 Split Directive (.SPLIT)

```
[label]        .SPLIT                                        [;comments]
```

This directive identifies to the assembler that the microprocessor this program is to be executed on has its base page in the range -128_{10} to $+127_{10}$ instead of 0 to $+255_{10}$. The only addresses this affects are those that are absolute (those in a .ASECT region). Base sector values (those in a .BSECT region) are not affected by this command. The .SPLIT directive also sets the .NOBAS directive. To have the assembler generate pointers use the following code:

```
.SPLIT
.ASECT
.=          9          ;RESERVE INTERRUPT POINTERS
.POOL       20         ;LEAVE ROOM FOR 20 ASSEMBLER GENERATED
                       ;POINTERS
```

NOTE

The .SPLIT directive must be used at the beginning of a program, before any code.

5.3.3 End Directive (.END)

```
[label]        .END      [address]                          [;comments]
```

The .END directive signifies the physical end of the source program. The optional address in the operand field may be either a symbol or a constant, and indicates an execution address to the loader. In other words, it causes a branch to the address of the first executable instruction (entry point in contrast to load point) after the load is complete. The assembler requires a .END as the last source statement in a program.

Examples:

1. No branch required:

```
LAST:          .END
```

2. Jump to the entry point at X'00A9:

```
               .END    X'00A9
```

3. Jump to the entry point labeled START:

```
               .END    START
```

5.3.4 Program Section Directives (.ASECT, .BSECT, .TSECT)

[label] { .ASECT / .BSECT / .TSECT } [;comments]

The three section directives enable the user to create a program in sections, producing a load module that is absolute (.ASECT), base-sector-relative (.BSECT), or top-sector-relative (.TSECT), or a combination of the three.

The .ASECT directive indicates to the assembler that the program statements which follow should be given absolute location values. The location values assigned by the assembler will be the actual addresses in memory where the assembled data will reside.

The .BSECT directive tells the assembler that the following statements should be given relative base-sector location values. It indicates that the assembled data in the following section of the program will be loaded into the base page, but the actual, physical base-page address will be defined at load time.

The .TSECT directive indicates the following program section should be given relative top-sector location values. Top sector is any area of memory outside the base sector (base page). .TSECT indicates the data will be loaded into some memory address outside the base-page, but the physical top-sector address will be specified at load time.

The programmer could change sector modes throughout the assembly, creating some sections of his program that are absolute, some that are base-page-relocatable and some that are top-sector-relocatable.

The assignment statement permits the value of the location counter for the current sector mode to be changed; for example:

.=.+5	may be used to increment the location counter by 5 (reserve 5 words of memory).
.=5	may be used to set the location counter to a specific value. (Notice this statement is absolute, whereas the statement above is relative. This statement is permissible only when in .ASECT mode.)

The assembler maintains separate location-counter addresses for each mode. At the beginning of each assembly, the addresses are set to zero, and the mode is initialized to .TSECT. A section directive is in affect until a different section directive is encountered. At that point the location counter assumes the mode and value of the new section directive. For instance, in the example below, the section changes from .TSECT to .ASECT and then back to .TSECT. Note that the location counter for .TSECT continues where it left off. If another .ASECT were encountered, it would start counting at 0121, the next location for the .ASECT counter.

Example:

```
                .TSECT
0000
  :
  :
001D
                .ASECT
                .=0100
0100
  :
  :
0120
                .TSECT
001E
  :
  :
```

NOTE

The assignment statement is used to change the location counter value. (See 5.2.)

5.3.5 List Directive (. LIST)

[label] . LIST immediate [;comments]

The . LIST directive controls listing of the source program. This includes listing in general, listing of unassembled code caused by the . IF and . IFC directives, listing of macro expansions, and listing of code generated by macro expansions.

Control of the various list options depends upon the state of the five least significant bits of the evaluated expression in the operand field. Table 5-2 shows the options available in the order of their priority with default values indicated by asterisks.

Table 5-2. List Options

Function	Bit	Value	Description
Master List Control	0	1 0	* Full listing. Suppress all listing.
. IF List Control	1	1 0	Full listing (of . IF's and . IFC's.) * Suppress unassembled code.
Macro List Control	2,3	11 10 00	List all code expanded during macro calls. List only code generated by macro calls. * List only macro calls.
Binary List Control	4	1 0	* List all the binary output by statements generating more than one word (for example, . ASCII). List only the first two bytes of generated data.

* default

Options are usually combined to give the desired type of listing. Some examples follow.

1. Full listing:

```
. LIST   01
```

2. Full listing and list all code expanded during macro calls:

```
. LIST   0D
    or  ┌──────────────  Full list
. LIST   01 ! 0C ──────  list all code expanded during macro calls
```

3. Suppress listing:

```
. LIST   0
```

5.3.6 Space Directive (.SPACE)

```
[label]        .SPACE      immediate                    [;comments]
```

The .SPACE directive skips forward a specified number of lines on the program listing.

Examples:

1. Skip 20 (decimal) lines:

```
.SPACE   20
```

2. Skip 20 (hexadecimal) lines:

```
.SPACE   020
    or
.SPACE   X'20
```

5.3.7 Page Directive (.PAGE)

```
[label]        .PAGE       [string]                     [;comments]
```

The .PAGE directive spaces forward to the top of the next page on the program listing. The optional string is printed as a page title on each page until a new string is encountered. No action is taken (except for a new page title) if the .PAGE directive is issued immediately after an assembler generated top-of-page request.

Example:

```
.PAGE   'TTY I/O ROUTINES'
```

5.3.8 Word Directive (.WORD)

```
[label]        .WORD       expression[,expression,...expression]    [;comments]
```

The .WORD directive generates 16-bit or 8-bit data words (depending on .DLEN directive) in successive memory locations. Each expression in a .WORD directive is evaluated, and the expression value is placed in the next available memory location.

If a label appears in the label field of a .WORD directive statement, the label refers to the memory location address of the first value.

In all expressions appearing in a single .WORD directive statement, the special symbol '.' has the value of the location counter that corresponds to the initial location.

5.3.9 ASCII Directive (.ASCII)

```
[label]        .ASCII      string[,string,...string]    [;comments]
```

The .ASCII directive stores data in successive memory locations by translating the characters in the string into their 7-bit ASCII equivalent code (see appendix A). Each string must be enclosed in single quote marks ('). Each character occupies one 8-bit byte in memory with the most significant bit set to zero. For 16-bit data, the first character in each string is placed in the high-order byte of the next available memory location. The second character is placed in the low-order byte. The third character is placed in the high-order byte of the next word, and so on. If there is an odd number of characters in a string, the last low-order byte is filled with a blank code (X'20). For the 8-bit data length, one character occupies one memory location with the most significant bit set to zero.

The .ASCII directive is used primarily to generate messages for output on Teletype or line printer.

Example:

```
.ASCII   'INPUT VALUE OF X'
```

5.3.10 Set Directive (.SET)

```
[label]        .SET        symbol,expression                    [;comments]
```

The .SET directive is used to assign values to reassignable (set) variables. A variable assigned a value with the .SET directive can be reassigned different values an arbitrary number of times.

Examples:

```
.SET    A,100                   ;SET A = 100
.SET    B,50                    ;SET B = 50
.SET    C,A-25*B/4              ;SET C = A - 25 * B/4
```

NOTE

The expression is always evaluated from left to right regardless of the operators used between the variables and constants.

5.3.11 Conditional Assembly Directives

```
[label]        .IF         expression                           [;comments]
               .ELSE                                            [;comments]
               .ENDIF                                           [;comments]
```

The conditional assembly directives selectively assemble portions of a source program based on the value of the expression in the .IF directive statement.

All source statements between a .IF directive and its associated .ENDIF are defined as a .IF-.ENDIF block. These blocks can be nested to a depth of ten. The .ELSE directive can be included optionally in a .IF-.ENDIF block. The .ELSE directive segments the block into two parts. The first part of the source statement block is assembled if the .IF expression is greater than zero; otherwise, the second part is assembled. When the .ELSE directive is not included in a block, the block is assembled only if the .IF expression is greater than zero. If an error is detected in the expression, the assembler assumes a true value (greater than zero).

Examples:

1. Two-part conditional assembly

```
.IF                 COMPR
. .
  .
  .                                  Assembled if COMPR greater than zero.
.ELSE
  .
  .                                  Assembled if COMPR less than or equal to zero.
.ENDIF
```

2. Nested .IF-.ENDIF block conditional assembly

```
.IF                 SMT
  .
  .                                  Assembled if SMT is greater than 0
.ELSE
  .
  .
.IF                 OBR
  .                                  Assembled if SMT is less than or equal to zero.
  .                                  zero.
.ENDIF
  .
  .
.ENDIF
```

Assembled if OBR is greater than zero and SMT is less than or equal to zero.

Labels appearing on .IF statements are assigned the address of the next assembled instruction. Labels cannot be used on .ELSE or .ENDIF statements.

Listing of unassembled code may be controlled by appropriate use of .LIST directives. (See 5.3.5.)

5.3.12 Global Directive (.GLOBL)

```
[label]        .GLOBL    symbol[,symbol,...symbol]        [;comments]
```

The .GLOBL directive lists a set of symbols as being global to all load modules that are linked and loaded together. This is the mechanism by which individually assembled programs can communicate with one another.

Each symbol in the operand field is marked by the assembler as a global symbol. If the symbol is within the current assembly, it may be referred to by other load modules. If the symbol is not within the current assembly, it is assumed to be defined in another assembly, and references to this symbol will be established at load time.

Any number of .GLOBL directive statements may occur within a single assembly. They are treated as a single .GLOBL directive statement with a large number of symbol operands.

Example:

```
        .GLOBL   PARM1,PARM2,SFLG,EFLG
```

5.3.13 Local Directive (.LOCAL)

```
[label]        .LOCAL                                    [;comments]
```

The .LOCAL directive establishes a new program region for local symbols (symbols beginning with a dollar sign ($)). Designated symbols between two .LOCAL directive statements have the meaning assigned to them only within that particular region of the program. A .LOCAL directive is assumed at the beginning and end of a program; thus, one .LOCAL directive in the module splits the program into two regions.

If the first character of a symbol is a dollar sign ($), the assembler attaches a unique character to the end of the symbol. Initially, this character is the exclamation point (!). Each time a .LOCAL is encountered, the value of the added character is advanced by one with the letter "Z" being the last legal value. Therefore, up to 58 local symbols can appear in one program. (See appendix A.)

Example:

```
                .LOCAL
$CLEAR:         LI    0,0
                JMP   $STORE
$STORE:         ST    1,SAVE1
                 .
                 .
                 .
                .LOCAL
$CLEAR:         LI    1,0
                JMP   $CLEAR
$STORE:         ST    0,SAVE0
```

5.3.14 Data Length Directive (.DLEN)

```
[label]        .DLEN        expression                    [;comments]
```

The .DLEN directive specifies the data length of a region in assembly.

The expression must evaluate to a value of zero or one. Zero specifies 16-bit data length while one specifies 8-bit data length. If .DLEN is not used in the assembly, the default data length is 16 bits.

Example:

```
                    .DLEN 1
                      .
                      .
8-bit data region     .
                      .
                      .
                    .DLEN 0
                      .
                      .
16-bit data region    .
                      .
                      .
```

Instruction statements may not appear in an 8-bit data section. The .WORD and .ASCII directives are used to specify data in 8-bit sections.

5.3.15 Pointer Directive (.PTR)

```
[label]        .PTR         expression[,expression,...]   [;comments]
```

.PTR generates a word for each expression (the same as .WORD), but furthermore each expression is implicitly a pointer or literal that will be automatically used if within addressing range of any instruction needing such a pointer or literal.

Example:

```
        .PTR   1,01000,SYM+20
```

5.3.16 Pool Directive (.POOL)

```
[label]        .POOL        immediate                     [;comments]
```

The .POOL directive allocates from 1 to 36 words of pool memory. If a pointer or literal is needed, the assembler will generate the pointer or literal in a nearby pool if within range and if no pool is available within range the assembler will generate a base-page pointer.

Example:

```
        .POOL  10                    ;ALLOCATE 10 POINTERS
```

5.3.17 No Base Directive (.NOBAS)

```
[label]        .NOBAS                                     [;comments]
```

The .NOBAS directive causes assembler-generated base-page pointers to be flagged with a warning error message. The message is printed only when the pointer is generated and is not printed if the same pointer is subsequently used for another instruction.

Chapter 6

MACROS

Programming in simple assembly language enables a user to be as efficient with his microprocessor resources as his capabilities allow. With assembly language, the user can specify explicitly every detail of the program operation; indeed, he must specify every detail. Because of this, a program in assembly language often takes longer to write than the same program written in a high-level language that fills in many details automatically according to its internal design. This design may or may not be compatible with either the machine the language operates on or the user's problem. Ideally, the user would like a programming language that will be close to the machine when it needs to be while remaining as natural as possible for the expression of his particular problem. The language should fill in details whenever they are routine and should leave the user free to specify the details whenever they are crucial. This ideal can often be closely approximated by the use of a versatile programming tool known as macros.

Macros are a form of text replacement that provide an automatic code-generation capability completely under the user's control. With macros, a user can gradually build a library tailored to his application, and with a library of macros oriented toward a particular application, a user who is not a software expert can produce efficient machine language code; and an experienced user can significantly reduce his program development time.

6.1 BASIC MACRO CONCEPTS

The main use of macros is to insert assembly language statements into a source program, as shown in figure 6-1. In the example, the original source program contains a macro instruction, or macro call, named WRITE. WRITE is a macro that writes a message. When the assembler processes WRITE, it inserts the predefined sequence of assembly language from the macro definition named WRITE into the source program immediately after the point of call. The process of inserting the text of the macro definition into the source program is called macro expansion. The expanded macro then is processed as if it were part of the original source program. You will note that the macro call itself does not produce any maching-language code. The directives used to define the limits of the macro definition (.MACRO and .ENDM) are explained in detail later in this chapter.

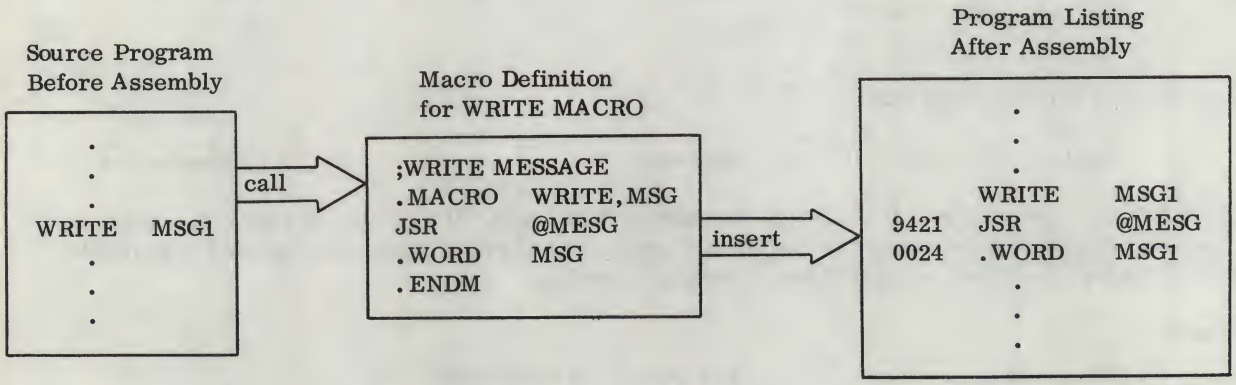


Figure 6-1. Statement Insertion

6.2 DEFINING A MACRO

Defining a macro means preparing the statements that constitute a macro definition. To define a macro, the following must be done:

- Give it a name.
- Declare any parameters to be used.
- Write the statements it contains.
- Establish its boundaries.

The following form is used to define a macro:

```
.MACRO      mname       [parameters]
              :
            macro body
              :
.ENDM
```

where:

(1) .MACRO is the directive that initiates the macro definition. Macros must be defined before their use. It is legal to define a macro with the same name as an already existing macro. The latest definition is always the operative one, but previous definitions are not discarded. They may be reactivated by using the .MDEL directive to delete the last macro definition (see 6.7.2).

(2) mname is the name of the macro (the name used to "call" the macro). The macro name must adhere to all rules for symbols (see 3.3.3.2).

(3) Parameters is the optional list of parameters used in the macro definition. The list of parameters must adhere to all the rules for expressions (see 3.3.4). The following are examples of legal and illegal . MACRO directives:

Legal		Illegal		Reason Illegal
.MACRO	MAC,A,B	.MACRO	SUB,1	Last character in parameter illegal
.MACRO	$ADD,OP1,OP2	.MACRO	1MAC,C,D	First character in macro name illegal
.MACRO	LIST,$1	.MACRO	MACB,25	First character in parameter illegal
.MACRO	MSG3	.MACRO	M$AC	Special character in macro name

(4) Macro body consists of assembly language statements. The macro body may contain simple text, text with parameters, and macro-time operators.

(5) .ENDM is the directive that terminates the macro definition.

The simplest form of a macro definition is one in which there are no parameters nor macro operators. The macro body is simply a sequence of assembly-language statements that are substituted for each macro call. The following shows how a simple macro can be defined to generate a delay loop.

```
        .MACRO   DELAY
        LI       AC0,07F
LOOP:   AISZ     AC0,-1
        JMP      LOOP
        .ENDM
```

6.3 CALLING A MACRO

Once a macro has been defined, it then may be called. A macro is called by placing the macro name in the operation field of an assembly language statement, and the parameters in the operand field. The following form is used for a macro call:

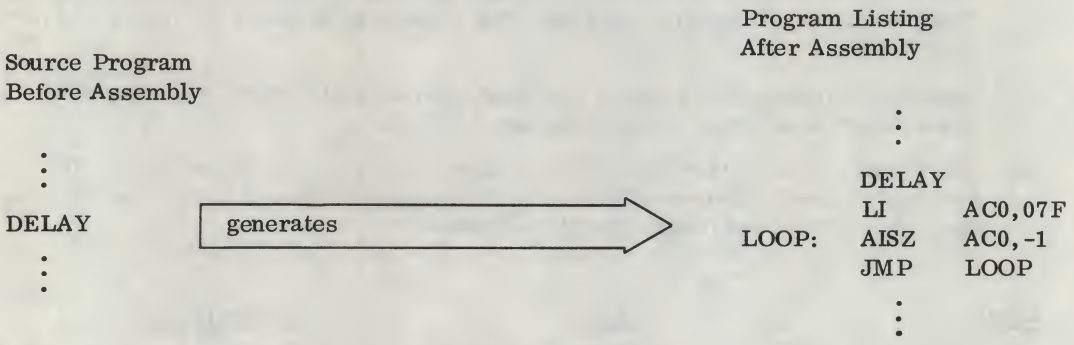
```
mname    [parameters]
```

where

(1) mname is the name previously assigned in the macro definition.

(2) Parameters is the list of input parameters. When a macro is defined without parameters, the parameter list is omitted from the call.

A call to the delay macro, defined in 6.2, would be as follows:



6.4 USING PARAMETERS

The power of a macro can be increased tremendously through the use of the optional parameters. The parameters allow variable values to be declared when the macro is called.

6.4.1 Macro Definition

The delay macro previously illustrated could be made more powerful through the use of parameters. By making the delay constants depend on the call, the same macro can be used for a variety of delays. In this example, DLYCON is a formal parameter and can be replaced with any value at expansion time.

```
;DELAY
        .MACRO   DELAY2,DLYCON
        LI       AC0,DLYCON
LOOP:   AISZ     AC0,-1
        JMP      LOOP
        .ENDM
```

Parameters need not be variables or numeric values, but can be any string. The following macro, for example, takes an ASCII string as input and generates a message string in memory suitable for input to the SC/MP firmware MESG routine.

```
        .MACRO    MSGSTR, LABEL, STRING
LABEL:  .ASCII    'STRING'
        .WORD     0
        .ENDM
```

The following macro generates a call to the MESG routine with the name of the message as input.

```
        .MACRO    MESG, MSGNAM
        JSR       07EA7
        .WORD     MSGNAM
        .ENDM
```

Note the principle here: the hexadecimal firmware address is maintained centrally in the MESG macro, not scattered all over the code as the macro calls will be.

6.4.2 Calling a Macro With Parameters

When parameters are included in a macro call, the following rules apply to the parameter list:

1. Commas or blanks delimit parameters.
2. Consecutive blanks are treated as a single delimiter.
3. A comma leading, following, or imbedded in a string of blanks is treated as a single delimiter.
4. A semicolon terminates the list and starts the comment field.
5. Quotes may be included as part of a parameter except as the first character of a parameter.
6. A parameter may be enclosed in quotes ('), in which case, the quotes are removed and the string is used as the parameter. This function is useful when blanks, commas, or semicolons are to be included in the parameter.
7. To include a quote in a quoted parameter, it must be entered as two consecutive quotes.
8. Missing or null parameters are treated as strings of length zero.

The following examples illustrate these rules.

NOTE

In these examples, for clarity, # indicates the parameter; strings are delimited by double primes (").

Macro Call	Parameter List	
MAC1 NAME NO ONE MAN	#1	"NAME"
	#2	"NO"
	#3	"ONE"
	#4	"MAN"
	#5,#6,...	" " (NULL PARAMETER)
MAC2 MADAM 'I''M' ADAM	#1	"MADAM"
	#2	"I'M"
	#3	"ADAM"
	#4,#5,...	" " (NULL PARAMETER)
MAC3 'A MAN','A PLAN','A CANAL' ; PANAMA		
	#1	"A MAN"
	#2	"A PLAN"
	#3	"A CANAL"
	#4,#5,...	" " (NULL PARAMETER)
MAC4 POOR,,'IS IN A DROOP'	#1	"POOR"
	#2	" " (NULL PARAMETER)
	#3	"IS IN A DROOP"
	#4,#5,...	" " (NULL PARAMETER)

The following examples show some macro calls to macros defined in section 6.2. These illustrate the parameter parsing and substitution.

Macro Call	Generated Code		
DELAY2 04F		LI	AC0,04F
	LOOP:	AISZ	AC0,-1
		JMP	LOOP
MSGSTR PROMPT 'ENTER VALUE'	PROMPT:	.ASCII	'ENTER VALUE'
		.WORD	0

6.4.3 Parameters Referenced by Number

6.4.3.1 '#' — Number of Parameters

'#' is a macro operator that references the parameter list in the macro call. When used in an expression, it is replaced by the number of parameters in the macro call. The following .IF directive, for example, causes the conditional code to be expanded if there are more than 10 parameters in the macro call:

```
.IF     #>10
```

6.4.3.2 '#N' — Nth Parameter

When used in conjunction with a constant or a variable, the '#' operator references individual parameters in the parameter list. The following example demonstrates how this function is used.

```
.MACRO  X
.WORD   #1,#2,#3
.ENDM
X       5,2,6      generates      .WORD   5,2,6
```

This relieves the need for naming each parameter in a long list, and allows powerful macros to be defined using arbitrary numbers of parameters.

6.4.4 Concatenation — '^'

The '^' macro operator is used for concatenation. When found, the '^' is removed from the output string and the strings on each side of the operator are compressed together after parameter substitution. If a set variable is used with the '^' operator, it is converted to a hexadecimal number before being placed in to the output stream.

Example:

```
.MACRO    IMAGINARY,X
R^X:  .WORD   0
I^X:  .WORD   0
.ENDM
```

Another example of the use of this operator is shown in 6.8.3.

6.5 LOCAL SYMBOLS

```
[label]          .MLOC          symbol[, symbol...]          [;comments]
```

When a label is defined within a macro, a duplicate definition results with the second and each subsequent call. The problem can be avoided by using the .MLOC directive to declare labels local to the macro definition.

Local symbols are replaced with unique names at expansion time with ZZxxxx, where xxxx is a 4-digit hexadecimal number. The user should avoid using his own labels of the above form as it may cause duplicate definition errors. The .MLOC directive may occur at any point in a macro definition, but it must precede the first occurrence of the symbols it declares local. If it does not, no error will be reported, but symbols used before the .MLOC will not be recognized as local.

6.6 CONDITIONAL EXPANSION

The versatility and the power of the macro assembler is enhanced by the conditional assembly directives. The conditional assembly directives (.IF, .ELSE, and .ENDIF) from chapter 4 allow the user to generate different lines of code from the same macro simply by varying the parameter values used in the macro calls. Three relational operators are provided:

= (equal)
< (less than)
> (greater than)

6.6.1 .IF Directive

When the macro assembler encounters a .IF directive within a macro expansion, it evaluates the relational operation that follows. If the expression is satisfied (evaluated greater than 0), the lines following the .IF are expanded until a .ELSE or a .ENDIF directive is encountered. If the expression is not satisfied (evaluated less than or equal to 0), only the lines from the .ELSE to the .ENDIF are expanded. See 4.5.11 for additional information on the conditional assembly directives.

The following macro for PACE simulates a BOC instruction that works on any register (rather than just AC0):

```
.MACRO    BOC2, REG, COND, LABEL
.IF       REG                    ;IF REG   0, MOVE TO 0
RCPY      REG, 0
.ENDIF
BOC       COND, LABEL
.ENDM
```

6.6.2 .IFC Directive

```
[label]          .IFC          string1  operator  string2          [;comments]
```

The .IFC directive allows conditional assembly based on character strings rather than the value of an expression as in the .IF directive. String1 and String2 are the character strings to be compared. Operator is the relational operator between the strings. Two operators are allowed: EQ (equal) and NE (not equal). If the relational operator is satisfied, the lines following the .IFC are assembled until a .ELSE or a .ENDIF is encountered. The .ELSE and .ENDIF directives have the same effect with the .IFC directive as they do with the .IF directive.

The primary application of the .IFC is to compare a parameter value such as #3 against a specific string. For example:

```
.IFC      #3 NE INTEGER
```

6.7 USEFUL DIRECTIVES

6.7.1 Set Directive (.SET)

```
[label]        .SET        symbol,expression        [;comments]
```

The set directive is used to assign values to symbols (variables). A variable assigned a value with the .SET directive can be reassigned different values an arbitrary number of times. Set variables are useful during macro expansion to control macro-time looping and macro communication. To insure value correspondence between pass one and pass two of the assembler, all values in the expression must be defined before use in a .SET directive. If a value is not previously defined, an error is reported and a value of zero is returned.

6.7.2 Macro Delete Directive (.MDEL)

```
[label]        .MDEL        mname[,mname...]        [;comments]
```

The .MDEL directive deletes macro definitions from the macro definition table and frees the buffer space used by the definitions.

Example:

```
        .MDEL    DELAY,MAC1
```

6.7.3 Error Directive (.ERROR)

```
[label]        .ERROR        [string]        [;comments]
```

The .ERROR directive generates an error message and an assembly error that is included in the error count at the end of the program. The directive is useful for parameter checking in macros.

The following example shows a rewrite of the BOC2 macro that checks that the input register is between 0 and 3 and reports an error if it is not.

```
.MACRO    BOC2,REG,COND,LABEL
.IF       (REG > -1) & (REG < 4)
.IF       REG > 0
RCPY      REG,0
.ENDIF
BOC       COND,LABEL
.ELSE
.ERROR    'ILLEGAL REGISTER VALUE'
.ENDIF
.ENDM
```

6.8 MACRO-TIME LOOPING

6.8.1 .DO and .ENDDO Directives

```
[label]        .DO    count        [;comments]
[label]        .ENDDO              [;comments]
```

Macro-time looping is facilitated through the .DO and .ENDDO directives. These directives are used to delimit a block of statements that are repeatedly assembled. The number of times the block will be assembled is specified in the .DO directive. The format of a .DO-.ENDDO block is shown on the following page.

```
        .DO     count
                 .
                 .
                 .
        source
                 .
                 .
                 .
        .ENDDO
```

NOTE

.DO, .ENDDO, and .EXIT are defined only within a macro definition.

6.8.2 .EXIT Directive

```
[label]     .EXIT                                   [;comments]
```

Early termination of looping in a .DO-.ENDDO block can be effected with the .EXIT directive.

This directive causes the current loop to be completed and then causes assembly to continue with the statement following the .ENDDO.

6.8.3 Examples of Macro-Time Loops

The following examples show the use of the .DO, .ENDDO, and .EXIT directives. The macro CTAB generates a constant table from 0 to MAX where MAX is a parameter of the macro call. Each word has a label DX:, where X is the value of the data word.

```
            .MACRO    CTAB,MAX
            .SET      X,0
            .DO       MAX+1
D^X:        .BYTE     X
            .SET      X,X+1
            .ENDDO
            .ENDM
```

Now a call of the form:

```
            CTAB      10
```

Generates code equivalent to:

```
            .SET      X,0
D00:        .WORD     X
            .SET      X,X+1
D01:        .WORD     X
            .SET      X,X+1
D02:        .WORD     X
              .
              .
              .
            .SET      X,X+1
D09:        .WORD     X
            .SET      X,X+1
D0A:        .WORD     X
```

The macro MSGLST generates a call to the PACE firmware MESG routine with each of the parameters on the macro call. The parameter list may be any length.

```
.MACRO      MSGLST
.SET        X,0
.DO         -1              ;SET FOR INFINITE LOOPING
.SET        X,X+1
.IF         X > #           ;CHECK IF NUMBER OF TIMES THRU LOOP (X)
                            ;IS > NUMBER OF PARAMETERS CALLED BY
                            ;MSGLST
.EXIT                       ;YES
.ELSE
MESG        #X              ;NO, CALL MEGS MACRO
.ENDIF
.ENDDO
.ENDM
```

Now a call of the form

```
MSGLST      MSG1,MSG2
```

generates code equivalent to

```
JSR         07EA7
.WORD       MSG1
JSR         07EA7
.WORD       MSG2
```

NOTE

Care must be taken when writing macros that generate a variable number of data words through the use of the .IF or the .DO. If the operands on these directives are forward references, their values will change between pass 1 and pass 2 and the number of generated words may change. Should this be the case, all labels defined after the macro call that has changed values will generate numerous assembly errors of the form

ERROR DUP. DEF.

6.9 NESTED MACRO CALLS

Nested macro calls are allowed; that is, a macro definition may contain a call to another macro. When a macro call is encountered during macro expansion, the state of the macro currently being expanded is saved and expansion begins on the nested macro. Upon completing expansion of the nested macro, expansion of the original macro continues. Depth of nesting allowed will depend on the parameter list sizes, but, on the average, about 10 levels of nesting will be allowed.

The following shows a simple example of macro nesting. The first macro defines the common function of ensuring data in AC0, while the second macro uses this macro in generating a BOC instruction.

```
.MACRO   CHKZRO,REG              ;CHECK THAT DATA IS IN REG 0
.IF      REG > 0
RCPY     REG,0                   ;COPY DATA TO AC0
.ENDIF
.ENDM

.MACRO   BOC2,REG,COND,DEST      ;SIMULATE BOC FOR ANY REG
CHKZRO   REG                     ;MAKE SURE DATA IN AC0
BOC      COND,DEST
.ENDM
```

A logical extension of a nested macro call is a recursive macro call, that is, a macro that calls itself. This is allowed, but care must be taken that an infinite loop is not generated.

6.10 NESTED MACRO DEFINITIONS

A macro definition can be nested within another macro. Such a macro is not defined until the outer macro is expanded and the nested .MACRO statement is executed. This allows the creation of special-purpose macros based on the outer macros parameters and, when used with the .MDEL directive, allows a macro to be defined only within the range of the macro that uses it.

Chapter 7

PROGRAMMING TECHNIQUES

This chapter discusses the programming techniques used to produce efficient PACE object code. Examples of coding are included to illustrate the method by which the techniques are implemented.

7.1 STACK

The hardware stack in PACE is used primarily for temporary storage of the contents of the Program Counter and the Status Flags Register during subroutine and interrupt service routine execution. The last-in/first-out accessing of the stack ensures that nested routines are exited in the reverse order of entry. The stack may also be used for data storage when data words can be sequentially stored in the reverse order required for retrieval.

A stack interrupt (interrupt level 1) is provided to handle a stack-full or a stack-empty condition. An interrupt is generated when the stack contains a single word and a read-stack operation occurs; this empties the stack. An interrupt is also generated when the stack contains eight words and a write-stack operation occurs; this fills the ninth word and leaves one word available to the interrupt. If the master interrupt enable (IEN) is zero, stack interrupts are ignored.

When a stack interrupt occurs, the condition of the stack can be tested by using the Branch On Condition (BOC) Instruction. Branch condition zero is set whenever the stack contains nine or more words.

NOTE

If a stack-pull (push) operation causes the stack to become empty (full) and IEN1 is '1', an interrupt request occurs. If this interrupt is not serviced and cleared (for example, because the master interrupt enable, IEN, is '0'), the interrupt is not cleared by subsequent stack-push (pull) operations. Thus, care must be exercised in using the stack when IEN is '0'.

The stack is implemented by using a file of 10 registers and a stack pointer. When the last word of the stack is written (that is, the stack is full), the pointer remains at the last register and all subsequent stack-push operations write data in the last register; thus, only the most recent data are retained.

When the stack becomes empty, the pointer addresses a nonexistent register, thereby causing a data word of all ones to be returned for all subsequent pull operations.

Use of the Stack Interrupt allows convenient extension of the stack into external memory for cases where 10 words of storage are not enough. When the stack is extended into external memory, Stack-full and Stack-empty Interrupts can be used to enter a routine that dumps a full stack into external memory and restores an empty stack from external memory. Thus, the effective size of the stack can be as large as the external memory. If the stack is not extended into external memory, the Stack Interrupts can be inhibited by turning off status flag IE1. In cases where a Stack-empty Interrupt is not desired (when both software and hardware stacks are empty), a dummy word may be pushed onto the stack during initialization.

The Stack Service Routine listed in figure 7-1 pushes four words onto a software stack when the hardware stack is full, and pulls four words off of a software stack when the hardware stack is empty.

NOTE

At least one word should always be left on the hardware stack by the Stack Service Routine to prevent a stack-empty interrupt from occuring after pushing the software stack. Similarly, only eight words should be pushed onto the hardware stack to prevent a stack-full interrupt.

The Stack Service Routine does not check for software stack overflow or underflow.

```
PACE ASSEMBLER REV-A      18 NOV 76                PAGE    1
STKINT SOFTWARE STACK ROUTIN


    1                           .TITLE  STKINT,' SOFTWARE STACK ROUTINES'
    2                   ;
    3                   ; THIS PROGRAM CONTAINS ALL ROUTINES NECESSARY TO SET
    4                   ; AND MAINTAIN A SOFTWARE STACK.
    5                   ;
    6 0000                      .ASECT
    7 0000                      .=2
    8 0002 0000 T               .WORD   STKINT
    9                   ;
   10 0000                      .TSECT
   11                   ;
   12                           .GLOBL  CLRSTK
   13        0000 A AC0         =       0
   14        0001 A AC1         =       1
   15        0002 A AC2         =       2
   16        0003 A AC3         =       3
   17                   ;
   18        0000 A STKFUL      =       0
   19        0001 A IEN1        =       1
   20        0009 A IEN         =       9
   21                   ;
   22                   ;  THE "STKINT" ROUTINE MAINTAINS THE SOFTWARE STACK.
   23                   ;
   24 0000 D165 A STKINT: ST       AC0,REG          ; SAVE ACCUMULATORS
   25 0001 D565 A         ST       AC1,REG+1
   26 0002 D965 A         ST       AC2,REG+2
   27 0003 6400 A         PULL     AC0              ; SAVE INTERRUPT RETURN ADDR
   28 0004 D13F A         ST       AC0,RETADR
   29 0005 4014 A         BOC      STKFUL,$FULL     ; TEST IF STACK IS FULL/EMPTY
   30 0006 6000 A         PUSH     AC0              ; TEST IF STACK IS ALMOST
   31 0007 400F A         BOC      STKFUL,$AFULL    ;  FULL
   32 0008 6400 A         PULL     AC0
   33                   ;
   34 0009 5104 A         LI       AC1,4            ; STACK IS EMPTY
   35 000A AD3A A $EMPTY: DSZ      STKPTR           ; RESTORE 4 WORDS ONTO STACK
   36 000B A139 A         LD       AC0,@STKPTR
   37 000C 6000 A         PUSH     AC0
   38 000D 79FF A         AISZ     AC1,-1
   39 000E 19FB A         .JMP     $EMPTY
   40                   ;
   41 000F C134 A $REST:  LD       AC0,RETADR       ; RESTORE INTR RETURN ADDRESS
   42 0010 6000 A         PUSH     AC0
   43 0011 C154 A         LD       AC0,REG          ; RESTORE PROGRAM'S STATUS
   44 0012 C554 A         LD       AC1,REG+1
   45 0013 C954 A         LD       AC2,REG+2
```

Figure 7-1. Stack Service Routine

```
46 0014 3100 A          PFLG    IEN1            ; CLEAR INTERRUPT LATCH
47 0015 3180 A          SFLG    IEN1
48 0016 7C00 A          RTI                     ; RESUME PROGRAM EXECUTION
49                ;
50 0017 6400 A $AFULL:  PULL    AC0             ; STACK IS ALMOST FULL
51 0018 5104 A          LI      AC1,4           ; NUMBER OF WORDS TO SAVE
52 0019 1901 A          JMP     .+2
53                ;
54 001A 5105 A $FULL:   LI      AC1,5           ; NUMBER OF WORDS TO SAVE
55 001B D522 A          ST      AC1,$TEMP
56 001C C920 A          LD      AC2,$ADR
57 001D 6400 A $LP1:    PULL    AC0             ; SAVE TOP WORDS OF STACK
58 001E D200 A          ST      AC0,(AC2)
59 001F 7A01 A          AISZ    AC2,1
60 0020 79FF A          AISZ    AC1,-1
61 0021 19FB A          JMP     $LP1
62                ;
63 0022 5104 A          LI      AC1,4
64 0023 6400 A $LP2:    PULL    AC0             ; SAVE LAST FOUR WORDS ONTO
65 0024 B120 A          ST      AC0,@STKPTR     ; SOFTWARE STACK
66 0025 8D1F A          ISZ     STKPTR
67 0026 79FF A          AISZ    AC1,-1
68 0027 19FB A          JMP     $LP2
69                ;
70 0028 C515 A          LD      AC1,$TEMP
71 0029 7AFF A $LP3:    AISZ    AC2,-1          ; RESTORE TOP WORDS OF STACK
72 002A C200 A          LD      AC0,(AC2)
73 002B 6000 A          PUSH    AC0
74 002C 79FF A          AISZ    AC1,-1
75 002D 19FB A          JMP     $LP3
76 002E 19E0 A          JMP     $REST           ; RESTORE STATUS AND RESUME
77                ;
78                ;  THIS ROUTINE WILL CLEAR BOTH THE HARDWARE AND SOFTWARE STA
79                ;
80 002F 3900 A CLRSTK:  PFLG    IEN             ; PREVENT FURTHUR INTERRUPTS
81 0030 49FE A          BOC     IEN,.-1
82 0031 6600 A          PULL    AC2             ; SAVE RETURN ADDRESS
83 0032 5109 A          LI      AC1,9
84 0033 6400 A          PULL    AC0             ; CLEAR HARDWARE STACK
85 0034 79FF A          AISZ    AC1,-1
86 0035 19FD A          JMP     .-2
87 0036 C133 A          LD      AC0,STKADR      ; CLEAR SOFTWARE STACK
88 0037 D10D A          ST      AC0,STKPTR
89 0038 3100 A          PFLG    IEN1
90 0039 3180 A          SFLG    IEN1            ; RESTORE INTERRUPT STATUS
91 003A 3980 A          SFLG    IEN
92 003B 6100 A          PUSH    AC1             ; DUMMY WORD
93 003C 1A00 A          JMP     (AC2)           ; RESUME PROGRAM EXECUTION
94                ;
95                ;  TEMPORARY DATA STORAGE
96                ;
97 003D         $ADR:   .=.+1
98 003E         $TEMP:  .=.+1
99 003F         $STAK:  .=.+5
100 0044        RETADR: .=.+1
101 0045        STKPTR: .=.+1
102 0046        SSTACK: .=.+X'20
103 0066        REG:    .=.+4
104 006A 0046 T STKADR: .WORD   SSTACK
105               ;
106       0000 A        .END
```

Figure 7-1. Stack Service Routine (Continued)

```
AC0     0000  A     AC1     0001  A     AC2     0002  A     AC3     0003  A*
CLRSTK  002F  GT*   IEN     0009  A     IEN1    0001  A     REG     0066  T
RETADR  0044  T     SSTACK  0046  T     STKADR  006A  T     STKFUL  0000  A
STKINT  0000  T     STKPTR  0045  T     $ADR    003D  T     $AFUL   0017  T
$EMPT   000A  T     $FULL   001A  T     $LP1    001D  T     $LP2    0023  T
$LP3    0029  T     $REST   000F  T     $STAK   003F  T*    $TEMP   003E  T

 NO ERROR LINES
 SOURCE CHECKSUM =4E43

INPUT FILE     2:STKINT.SRC ON HSA 006
```

Figure 7-1. Stack Service Routine (Continued)

7.2 SUBROUTINES

Subroutines are an invaluable tool in computer programming. Basically, a subroutine is a segment of code outside the normal program flow that can be executed at several places in the program through use of a JSR (Jump to Subroutine) Statement. The label on the first statement names the subroutine and serves as its entry point. The last statement executed in a subroutine must be an RTS (Return from Subroutine), which serves as the exit. Consider the following simple subroutines: SAVREG, which stores the four general-purpose registers in the stack; GETREG, which restores the registers from the stack; and the calls to these subroutines.

```
SAVREG:  XCHRS    0
         PUSH     1
         PUSH     2
         PUSH     3
         PUSH     0
         RTS

GETREG:  PULL     0
         PULL     3
         PULL     2
         PULL     1
         XCHRS    0
         RTS
           .
           .
           .
         JSR      SAVREG          ;SAVE REGISTERS
           .
           .
           .
         JSR      GETREG          ;RESTORE REGISTERS
           .
           .
           .
```

The use of subroutines has several advantages. First, subroutines can divide a complex program into a number of discrete parts. These then can be tackled individually and programmed with relative ease. If several programmers are working together on a program, subroutines provide a logical and efficient way to divide the labor. Local variable blocks, in which variables starting with a '$' have meaning only within that block can be used to enhance subroutine compatibility and are especially useful in routines that can be used in several programs.

A second advantage of using subroutines is reduced memory requirements. Input/output functions, for example, are often rather large routines that can be written once and called from several places in the program. Without this facility, programs would quickly grow to unmanageable size.

Occasionally, a subroutine requires no input and generates no output, as in the example above. Usually, however, the subroutine needs to pass information between itself and the calling program. One way to handle such communication is to use the four general-purpose registers and the stack to hold the values being passed. Another method is to use a common block of RAM (Random Access Read/Write Memory) to store data and results. Finally, the addresses of the data or results may be stored in the registers, stack, or memory and the subroutine may fetch the data itself.

7.3 INPUT AND OUTPUT PROGRAMMING TECHNIQUES

The programming of data transfers between read/write memory and peripheral devices generally is classified as input/output programming. Depending on the significance of the input/output operations in the overall program, different approaches to input/output program implementation are recommended; these approaches are described in the following sections.

7.3.1 Programmed Input/Output

A programmed input/output operation is initiated and completed under the control of the initiating program. In figure 7-2, the program being executed starts the input/output operation; then, the program waits for the operation to be completed before continuing.

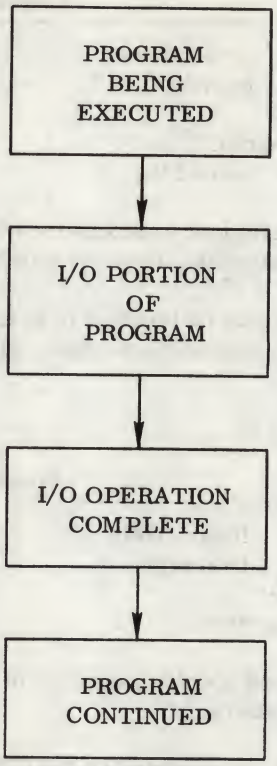


Figure 7-2. Programmed Input/Output

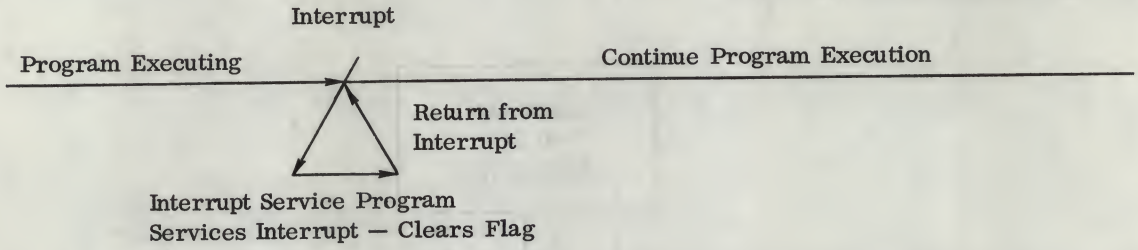
PACE allows any memory-reference instruction to execute a programmed input/output operation. Peripheral device controllers are assigned specific memory addresses, any of which, when referenced by a memory-reference instruction, executes the input/output operation. It is necessary that the memory addresses assigned to the peripheral device be unique to the device; that is, no other peripheral device uses the same assigned memory addresses, nor is there memory with the same addresses. Also, the device controller must contain the necessary logic to decode its assigned addresses, and, then, to gate data on and off the data bus. For example, memory addresses X'7FFF and below might be reserved for read/write or read-only memory and memory addresses X'8000 to X'FFFF might be reserved for peripheral device controllers. The user, however, can define his own convention.

The actual program steps required to enable programmed input/output depend on the design of the device controller.

7.3.2 Interrupt Input/Output

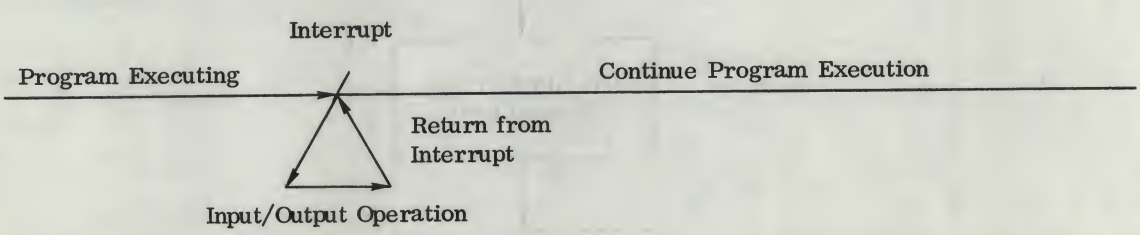
In certain cases, an input/output operation initiated by a program requires a significant length of time (many milliseconds) for execution; during this time, the program might perform other tasks. In other cases, the frequency of input/output service that requires the use of a certain input/output device might be such that it would be convenient for the program to ignore the device unless it specifically requires service. Each of these situations may be handled by taking advantage of the interrupt system and by employing interrupt input/output for devices that have interrupt capability.

In figure 7-3, the program might initiate the input/output operation as part of its normal sequence-of-operation and might set a flag indicating that such action was taken. An input/output device that has interrupt capability, upon completion of an input/output operation, transmits an interrupt to PACE to indicate completion of the operation.



The flag may be employed by the original program to determine whether or not the input/output operation has been completed and whether or not the input/output device is still busy.

Another way that an input/output operation may be initiated is to transmit an interrupt signal to the CPU. In this case, the program being executed is interrupted, the input/output operation is effected, and, then, the interrupted program is resumed.



Interrupt input/output requires a definite and specific sequence of events, irrespective of what peripheral device is to be serviced; the following sequence occurs.

1. In order for an interrupt to be accepted by the CPU, the master interrupt enable flag (IEN) must be enabled (set to 1), and for the PACE, the particular interrupt level corresponding to the device (IE0 through IE5) must be enabled also. If either flag is disabled (set to 0), interrupt signals from peripheral devices are rejected. The master interrupt enable may be enabled by the instruction

```
LABEL:    SFLG    9
```

and disabled by the instruction

```
LABEL:    PFLG    9
```

The interrupt level flags may be manipulated via the following sequence of instructions:

```
          PUSHF
          PULL    AC0
```

Code to modify flags

```
          PUSH    AC0
          PULLF
```

NOTE

Level 0 interrupts must be enabled via SFLG 15.

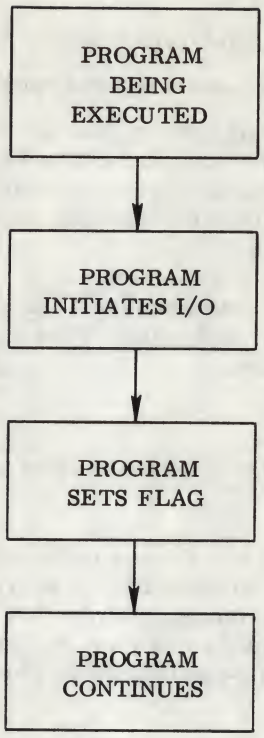


Figure 7-3. Interrupt Input/Output Initiation

2. Once an interrupt has been received and accepted by the CPU, the following steps occur automatically, and under control of the CPU.

   a. The instruction currently being executed is completed, and the memory address of the next instruction is pushed onto the stack.

   b. Interrupts are disabled (IEN is set to 0). Therefore, no further interrupts will be accepted by the CPU until interrupts are re-enabled.

   c. The priority encoder (see figure 2-5) provides an address that is used to access the interrupt pointer for the highest priority interrupt request.

3. The interrupt pointers are stored in locations 2 through 6 (see table 7-1) for interrupt requests 1 through 5, respectively. The interrupt pointer specifies the starting address of the interrupt service routine for the particular interrupt level, except for level 0. For level 0, location 7 is the PC-save address and the first instruction of the interrupt handler must reside in location 8. The interrupt service routine must perform a number of housekeeping tasks before the required input/output operation can proceed. Tasks, in order of normal execution follow.

   a. Save the contents of accumulators and status register so that they can be restored just prior to returning from the interrupt. Accumulator and status register contents can be saved on the stack, but since the stack has just 10 words, more commonly a data area in memory is set aside for temporary data storage.

   b. Determine the source of the interrupt. How this is done depends on the design of the peripheral device controllers, but usually controllers are designed to follow the interrupt request signal by transmitting a data bit (or word) that identifies the source of the interrupt.

   c. Once the interrupt has been identified, jump to the routine that services the identified device. This input/output service routine is written using programmed input/output as described in 7.3.1.

4. Execute the selected device's input/output service routine.

5. Restore to the accumulator and status register contents that were saved in step 3a.

6. For other than level 0 interrupts, return from the interrupt by executing an RTI Instruction. This re-enables interrupts by setting flag IEN to 1 and pulls the return address (saved in the stack by step 2a) into the program counter, so execution continues at the program instruction following the interrupt. For level 0 interrupts, SFLG 15 to re-enable interrupt level 0 and then jump indirect via location 7.

An example of an Interrupt Service Routine for interrupt level 3 is shown in table 7-2. Memory location 4 contains the address of the first instruction in the routine. When a level 3 interrupt occurs, the first instruction preserves the state of the flags on the stack.

NOTE

IEN is set to 0 by the interrupt prior to being saved on the stack.

The flag data then are loaded into AC0 and all bits that are to be modified are masked out to zero. The desired bits then are set to 1 by ORing with IESTAT. If the routine is interruptable, then IE3 is set to zero and IEN is set to one. The modified status word then is transferred from AC0 to the status register. The actual servicing of the interrupting device then takes place. At the end of the routine, the flags are restored and a return instruction is executed. If the interrupts are to be re-enabled, the RTI Instruction must be used since RTI sets IEN to 1 and restores the PC from the stack.

NOTE

Status register masking is necessary only when interrupt enable status is to be modified to allow higher-priority devices to interrupt. Pushing the status register onto the stack is necessary only if the routine alters the contents of the status register.

Table 7-1. Locations of Interrupt Pointers

Interrupt Pointer	Location
Interrupt 0 Program	8
Interrupt 0 PC	7
Interrupt 5	6
Interrupt 4	5
Interrupt 3	4
Interrupt 2	3
Interrupt 1	2
Not Assigned	1
Initialization Instruction	0

Table 7-2. Interrupt Service Routine Example

Assembly Code			Explanation
	.=4		Set location counter equal to 4.
	.WORD	ISERV3	Pointer to service routine.
	.=500		Set location counter equal to 500.
ISERV3:	PUSHF		Save flags on stack.
	CFR	AC0	Move flags to AC0.
	AND	AC0,MASK	Mask out old Interrupt Enable status.
	OR	AC0,IESTAT	OR in new Interrupt Enable status.
	CRF	AC0	Store in flag register.
	.		.
	.		.
	.		.
	.		Interrupt Service Routine.
	.		.
	.		.
	.		.
INTXIT:	PULLF		Restore flags.
	RTI		Return to interrupted routine.
MASK:	.WORD	(mask data)	
IESTAT:	.WORD	(Interrupt Enable status data)	

Processing following a "stack full" interrupt (level 1) requires essentially the same sequence of steps as outlined previously. The "stack full" interrupt must be identified by testing the "stack full" condition:

```
STKFUL      =       0
LABEL:      BOC     STKFUL,HERE
```

If the "stack full" condition exists, execution continues at HERE.

NOTE

Occurrence of a "stack full" interrupt may cause loss of data since it will result in the PC being pushed onto the stack.

7.3.3 Input/Output System Organization

Depending on the intended application for the PACE, the type of input/output programming described in section 7.3.1 may or may not be adequate.

In a dedicated application where the PACE is used as a controller, or will rarely be subject to extensive reprogramming, it is efficient to incorporate input/output programming steps into the body of the program.

When the PACE is to be constantly programmed, and particularly when peripheral devices are subject to change, it is more efficient to introduce the "logical unit" concept into input/output programming. Using this concept, programs are written to access peripheral devices functionally, rather than physically. For example,

Logical Unit 0 may be the operator interface device.
Logical Unit 1 may be the bulk output device.
Logical Unit 2 may be the bulk storage device.
Logical Unit 3 may be the data entry device.
Logical Units 4-7 may be data transmitting devices.

The operator interface device may be a teletypewriter keyboard, or a CRT (cathode ray tube) terminal. The bulk output device may be a line printer or a teletypewriter printer, or a paper tape punch.

The bulk storage device may be a magnetic disc, a magnetic tape, or a cassette unit.

The data entry device may be a teletypewriter keyboard, the teletypewriter paper tape reader or a high-speed paper tape reader.

The data transmitting devices may be analog-to-digital converters, intermediate magnetic storage devices, or specially wired external signal lines.

Input/Output programming now has three parts:

a. A generalized, logical-unit-oriented program to process requests for input/output.

b. A set of device drivers that link the logical unit requested in "a" above with the required physical unit.

c. Programs that actually enable the input/output operation.

7.3.3.1 Generalized Call to Input/Output

One subroutine will initiate all input/output operations with the exception of those operations that can only be initiated by an external interrupt. Let us call this subroutine IOS.

The execution of any input/output operation requested by a program will start with one common subroutine call

```
LABEL:    JSR     @IOS
          .WORD   LIST
```

where IOS provides, on the base page, the starting address of the input/output initiation subroutine, and LIST provides the memory address where the required input/output operation is defined. At LIST, the following information must be provided, using any convenient hexadecimal code:

1. The input/output operation to be performed should include:
   a. Read
   b. Write
   c. Open (Initialize flags, counters or other conditions, if needed).
   d. Close (Provide device use termination processing, if needed).
   e. Position to specified record and file.
   f. Backspace (or forward space) a set number of records and/or files.
   g. Return device status.
2. Input/output operation variables, including:
   a. ASCII or binary for data transfers.
   b. Echo or no echo for teletype.
   c. Formatted or unformatted for printed output.
3. Base address and length of memory buffer for read and write operations.
4. Record and file number for position and backspace/forward space.

The IOS subroutine will interpret the information provided at LIST, then call the device driver for the physical unit corresponding to the requested logical unit. IOS will contain a physical unit assignment table to link physical units to logical units. For example, if there are eight logical units and six physical units, the table may take the form:

```
PUTBLE:   .WORD    X'0000      ;LU 0 = PU 0
          .WORD    X'0100      ;LU 1 = PU 0
          .WORD    X'0201      ;LU 2 = PU 1
          .WORD    X'0302      ;LU 3 = PU 2
          .WORD    X'0404      ;LU 4 = PU 4
          .WORD    X'0503      ;LU 5 = PU 3
          .WORD    X'0605      ;LU 6 = PU 5
          .WORD    X'0705      ;LU 7 = PU 5
          .WORD    X'0806      ;LU 8 = PU 6
```

7.3.3.2 Device Drivers

The principal purpose of a device driver is to keep track of the status of a particular peripheral device during and between input/output calls. The device driver will maintain a device control block which is a data area dedicated to each peripheral device (one data area per device), where the following information is stored.

1. Busy/not busy. This serves two purposes.
   a. To selectively disable/enable individual peripheral devices.
   b. To selectively disable other peripheral devices during certain phases of this device's operation.
2. Record and file to which device was last positioned. The Open Call to IOS will reposition to this record and file, thus allowing reinitiation of discrete portions of input/output operations in the event of error conditions (bulk storage devices only).
3. Current record and file (bulk storage devices only).
4. Requested record and file (bulk storage devices only).
5. Selected parameters, coefficients, scale or conversion factors required or used by the device.
6. Condition of last operation: Successful, doubtful or error.

The device driver will now call the subroutine which executes the actual input/output operation.

7.4 8-BIT DATA LENGTH

When using the 8-bit data configuration, the 8-bit data is right justified in the 16-bit accumulator. The state of the leftmost 8 bits and the consequent effect on microprocessor operations must be considered. The following items are reviewed in the following paragraphs with respect to the 8-bit data length: data I/O, memory addressing, status flags, conditional branches, shifts and rotates, immediate instructions and mixed data lengths.

7.4.1 Data Input/Output

A system using the 8-bit data configuration usually has a 16-bit instruction memory (typically ROM), an 8-bit data memory, and an 8-bit peripheral device interface. When data is loaded into an accumulator from the 8-bit memory or peripheral device, the unused eight data lines may be driven to a logic '0' by the use of eight open collector gates. Thus, the left byte of the accumulator is zero. The unused eight data lines also may be left open (saving eight gates), in which case the left byte has an undetermined value dependent upon the system noise and previous states. In most cases, a nonzero left byte is of no concern, since status flags, conditional branches,

shifts, and rotates ignore the left byte. However, the programmer must be aware of the nonzero value, since the results of instructions such as Copy Register to Flags (CRF) are determined by the left byte as well as the right byte. In cases when the state of the left byte is significant, that byte may be set to zero by using the shift instructions with a count of zero.

7.4.2 Memory Addressing

Both the indexed and base-page addressing modes require some consideration when using the 8-bit data configuration. For base-page addressing, accessing both 16-bit (program words) and 8-bit (data words) data using the base-page mode may be desirable. Since two different memories are used, splitting the base page between the two memories also may be desirable. Base-page splitting is accomplished most easily by using the Base-Page Selection (BPS) input (see PACE Data Sheet) and the .SPLIT directive, 5.3.2, to cause the base-page address to be in the range of -128 to +127, rather than 0 to +255.

For indexed addressing, Accumulators 2 and 3 are used as 16-bit memory pointers. If Accumulators 2 and 3 are loaded from the 8-bit memory, the upper byte may be set equal to the sign of the lower byte by using the LSEX Instruction. Thus, a 16-bit signed twos-complement number results.

7.4.3 Status Flags

The Overflow and Carry Flags are modified by arithmetic instructions. If the 8-bit configuration is selected by the state of the Byte status flag, the Overflow and Carry Flags are set based on the lower 8-bit byte only. That is, the Carry Flag is set if there is a carry out of the lower byte and the Overflow Flag is set based on an arithmetic overflow of the lower byte.

The Link Flag is affected by shift and rotate instructions. The Link Flag is set by the data shifted out of the lower byte when the 8-bit configuration is selected.

7.4.4 Conditional Branches

The branch and skip instructions are modified to account for the 8-bit data length. The skip instructions (SKNE, SKG, SKAZ, ISZ and DSZ) test only the lower byte. Thus, if 8-bit accumulator data is compared with a 16-bit program memory word, the contents of the upper byte of both words are ignored. The Add Immediate, Skip if Zero Instruction (AISZ) is the only instruction that tests the entire 16-bit result when the 8-bit configuration is selected. Thus, the AISZ Instruction can be used to increment the index accumulators (AC2, AC3) without skipping every time the lower byte is zero. Consequently, the sign of 8-bit numbers must be extended (LSEX Instruction) to properly detect zero when using AISZ with 8-bit data.

7.4.5 Shifts and Rotates

The shift and rotate instruction group (SHL, SHR, ROL, ROR) operates on the lower byte only and sets the upper byte to zero. Shift instructions with a count of zero provide a convenient means of setting the left byte of accumulators to zero when 8-bit data is used.

7.4.6 Immediate Instructions

The immediate instructions (LI, CAI, AISZ) all provide 16-bit, twos-complement data inputs. When working with 8-bit data, the upper byte usually can be ignored. If required, the upper byte can be cleared using a shift instruction.

7.4.7 Mixed Data Lengths

Working with 8-bit data and 16-bit instructions sometimes necessitates performing arithmetic operations by using a 16-bit operand from the program memory and an 8-bit operand from the data memory. If the result is to be treated as 8-bit data, no special considerations are required. If the result is to be treated as 16-bit data, the sign of the 8-bit operand must first be extended by using the LSEX Instruction. Also, signals (carry, overflow, and conditional branches) that are only a function of the lower byte should not be used. Alternatively, the Data Length Flag may temporarily be set to 16 bits, if desired.

7.5 TEXT PROGRAMMING TECHNIQUES

When programs require extensive dialog, textual display, or printout, attention should be given to the technique that programs the textual printout, since it is likely to be subject to modification. One technique that readily lends itself to the PACE is the literal pool. A literal pool is an area within an assembled program where the literals used within the program are stored. Within the literal pool, all duplication of text is eliminated. For example, consider the following five messages:

1. ENTER 5 COEFFICIENTS
2. COEFFICIENT OUT OF ALLOWED RANGE. RE-ENTER
3. ANSWER =
4. ANOTHER?
5. NO VALID ANSWER. RE-ENTER 5 COEFFICIENTS

Text for the five messages may be stored in a literal pool as shown below.

```
2E52        L1:         .ASCII      '.RE-'
452D
454E        L2:         .ASCII      'ENTER'
5445
5220
3500        L3:         .ASCII      '5'
2C43        L4:         .ASCII      ' COEFFICIENT'
4F45
4646
4943
4945
4E54
5300        L5:         .ASCII      'S'
204F        L6:         .ASCII      ' OUT OF ALLOWED RANGE'
5554
204F
4620
414C
4C4F
5745
4420
5241
4E47
4500
414E        L7:         .ASCII      'ANOTHER?'
4F54
4845
523F
414E        L8:         .ASCII      'ANSWER'
5357
4552
```

```
3D20        L9:         .ASCII      '= '
4E4F        L10:        .ASCII      'NO VALID '
2056
414C
4944
2000
            L11:        .=.+1
```

Messages are created by indexing the literal pool using a repeating sequence of two words. Word 1 contains the displacement from the base of the literal pool to the first character to be printed. Word 2 contains the number of characters to be printed. An X'FFFF in the first word of a word pair indicates an end of message; otherwise, another segment of the message is sought in the next word pair. Each of the five messages described above could be created by the index sequence shown below.

```
0004        I1:         .WORD       L2-L1       ;ENTER 5 COEFFICIENTS
0014                    .WORD       L6-L2
FFFF                    .WORD       0FFFF
000B        I2:         .WORD       L4-L1       ;COEFFICIENT
000C                    .WORD       L5-L4
0018                    .WORD       L6-L1       ;OUT OF ALLOWED RANGE
0015                    .WORD       L7-L6
0000                    .WORD       L1-L1       ;.REENTER
000A                    .WORD       L3-L1
FFFF                    .WORD       0FFFF
0035        I3:         .WORD       L8-L1       ;ANSWER=
0008                    .WORD       L10-L8
FFFF                    .WORD       0FFFF
002D        I4:         .WORD       L7-L1       ;ANOTHER?
0008                    .WORD       L8-L7
FFFF                    .WORD       0FFFF
003D        I5:         .WORD       L10-L1      ;NO VALID ANSWER
0009                    .WORD       L11-L10
0035                    .WORD       L8-L1
0006                    .WORD       L9-L8
0000                    .WORD       L1-L1       ;.REENTER 5 COEFFICIENTS
0018                    .WORD       L6-L1
FFFF                    .WORD       0FFFF
```

A subroutine generates the printed messages. To write a message, the procedure is to call the subroutine and to specify the index that identifies the message to be printed. For example, to print "Answer=", the call would be as shown below.

```
            JSR         WRIT
            .WORD       I3
```

Subroutine WRIT calculates the section of the literal pool to be printed. The first character is at the address: L1 plus the contents of I3. The number of characters to be printed is derived from the contents of I3 + 1. The next byte contains X'FFFF, so printing is finished; otherwise, printing would continue with the next pair of index bytes specifying the next string of characters.

Chapter 8

ASSEMBLER INPUT/OUTPUT FORMATS

The information in this chapter is common to all PACE assemblers.

The PACE assembler programs accept free-format PACE assembly-language statements as source input. The programs interpret the statements and output a program listing and a machine-language load module. The load module may be loaded and executed by a PACE microprocessor. The load module may also be used to program ROMs or PROMs. Detailed operation instructions on the PACE assembler programs are contained in the installation and operating instructions for the assemblers.

Each PACE cross assembler requires the following minimum complement of peripherals: a source input unit, a program listing output unit, and a binary output unit. In addition, some assemblers require a scratch unit for multipass processing.

8.1 INPUT/OUTPUT FILES

The input and output files required by the assemblers are listed below and explained in the following paragraphs.

Function	File Format	Logical Record Length
Source File (Input)	Sequential	80 bytes
Program Listing File (Output)	Sequential	121 bytes
Load Module File (Output)	Binary	36 bytes

8.1.1 Source File (Input)

The Source File may be input via punched cards or paper tape or, in some cases, magnetic disc.

8.1.2 Program Listing File (Output)

The program listing file contains ANS carriage control characters. The format of the program listing written from this file follows.

Each line in the cross assembler program listing contains the following sequential columns: line number, location counter, value, indicator, source statement, and error message — defined as follows:

- Line number is the decimal line number of the source input statement. All source statements not deleted by conditional assembly directives are assigned sequential numbers.
- Location counter is the current hexadecimal value of the location counter. Any labels in the source statements are assigned this value.
- Value is the hexadecimal value of the code generated (or assignment made). For assembler statements that do not generate code, this field is blank.
- Indicator is a one-character symbol that describes the relocation characteristics of the code generated. See the operating instructions of your assembler for the definition of the symbols.
- Source statement is the reproduction of the source statement.
- Error message appears on the line(s) following the statement line if an error is detected. The question mark to the right of the error message designates, as closely as possible, the position of the error in the statement.

  Error messages are defined in the Installation and Operating Instructions of each assembler.

At the end of the program listing, a list of generated pointers is provided (if generated anywhere); a symbol table is produced; a message is printed noting the number of errors discovered by the assembler program; and the source and object checksums are printed.

8.1.3 Load Module (Output)

The Load Module (LM) file contains loading information and object code produced from the source statements. The LM file is an unformatted file composed of a sequence of records, each containing 18 words or less. The representation of the records depends on the storage medium. There are four types of LM records:

Title Record (one per LM file)
Symbol Record (variable number per LM file)
Data Record (variable number per LM file)
End Record (one per LM file)

The records are produced in the sequence shown in figure 8-1. Independent of the record type, the first two words in each record always have the same interpretation. The first word specifies the record type (bits 15 and 14) and the length of the record body (bits 13 through 0). The second word contains a checksum for error detection. The checksum is formed by taking the arithmetic sum (modulo 2^{16}) of all the bytes in the record body.

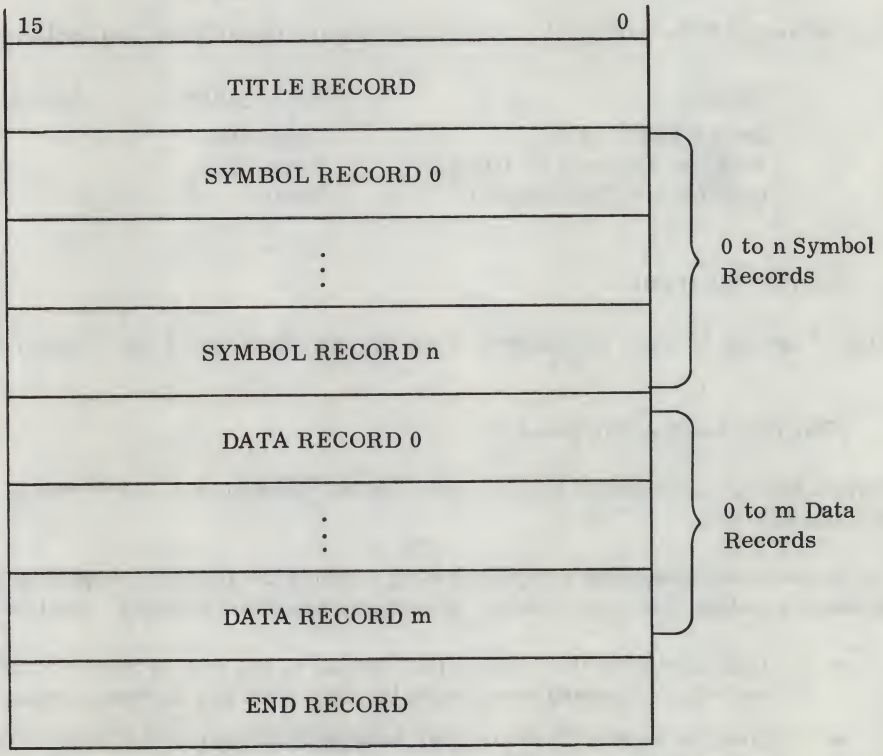
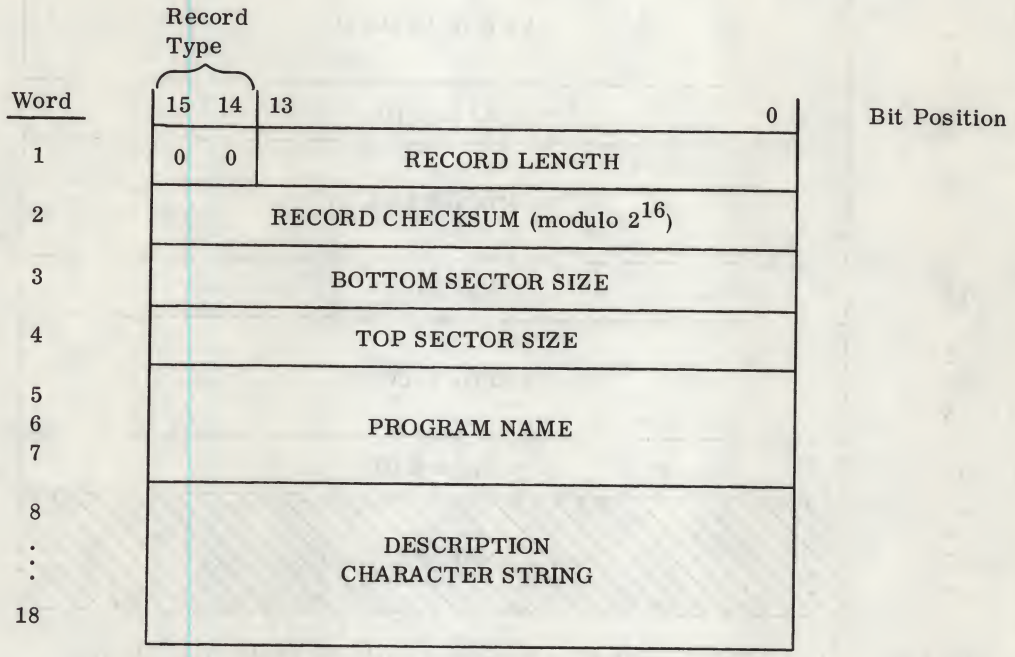


Figure 8-1. LM File Format

8.1.3.1 Title Records

The Title Record identifies the load module by name and, optionally, by a qualifying character string. These two identification items are supplied by the last .TITLE directive statement in the source program. If the .TITLE directive is not included, a default name (MAINPR) is used. If the default name is assigned, the qualifying character string is empty. Also included in the Title Record are two values that specify an estimate of the amount of storage utilized in the base sector and the top sector of memory. Determining the storage utilization is done by keeping track of the maximum value held by the two respective location counters. Figure 8-2 illustrates the format of the Title Record.



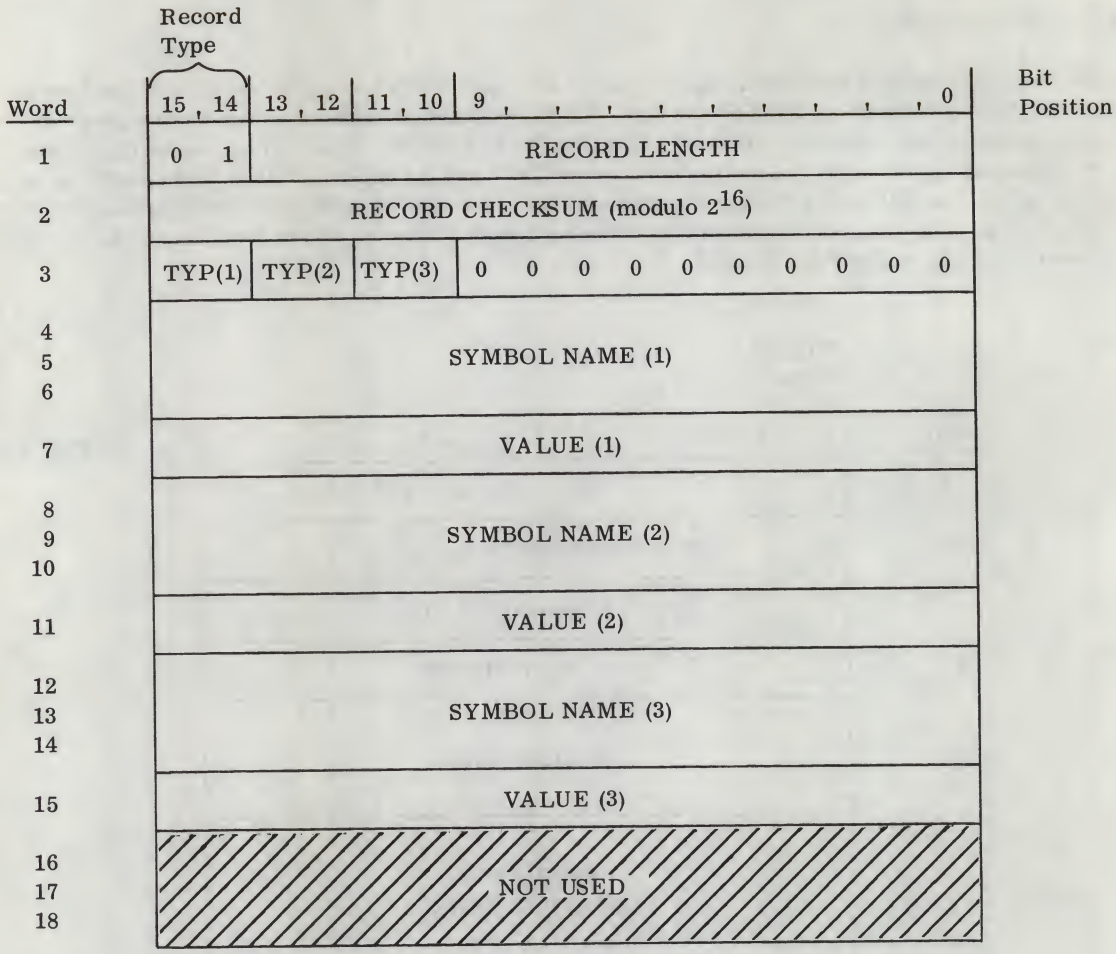
NOTES:

1. The program name and the descriptive string are composed of 7-bit ASCII characters. Both are right-justified with zeros-fill at the end.
2. Only the first 22 characters in the descriptive string (of the source statement) are used in the title record.

Figure 8-2. Title Record Format

8.1.3.2 Symbol Records

The Symbol Records specify values for global symbols that are internal to the current LM. The symbols then can be referenced by other LMs. In addition, global symbols that are external to the LM are specified with associated linkage information. Figure 8-3 illustrates the format of the Symbol Record.



NOTES:

1. TYP(i) specifies the relocation mode for SYMBOL NAME (i).

TYP(i)	Relocation Mode
0	Absolute
1	Base Sector
2	Top Sector
3	External

2. VALUE (i) is the absolute address (TYP(i) = 0), relocatable address (TYP(i) = 1 or 2), or external number (TYP(i) = 3).

3. SYMBOL NAME (i) is composed of 7-bit ASCII characters. If a symbol is less than six characters long, the remaining characters are zero.

Figure 8-3. Symbol Record Format

8.1.3.3 Data Records

The Data Records contain the actual data and the instruction words to be loaded into memory. Each Data Record contains the initial load address and the address mode for the first data word in the record. Subsequent data are loaded sequentially. Also, for each data word, there is a 2-bit field that specifies relocation information. Any time a discontinuity (that is, a change of sector or an empty block) exists in the data to be loaded, the current record is terminated (possibly with fewer than 12 data words) and a new record is initiated. Figure 8-4 illustrates the Data Record Format.

Word	Record Type 15, 14	13, 12	11, 10	9, 8	7, 6	5, 4	3, 2	1, 0	Bit Position
1	1 0	RECORD LENGTH							
2	RECORD CHECKSUM (modulo 2^{16})								
3	0 0	0 0	0 0	0 0	0 0	0 0	0 0	ATYP	
4	INITIAL LOAD ADDRESS								
5	TYP1	TYP2	TYP3	TYP4	TYP5	TYP6	TYP7	TYP8	
6	TYP9	TYP10	TYP11	TYP12	0 0	0 0	0 0	0 0	
7	DATA (1)								
⋮	⋮								
18	DATA (12)								

NOTES:

1. ATYP specifies the address mode for the INITIAL LOAD ADDRESS.

ATYP	Address Mode
0	Absolute
1	Base Sector Relocatable
2	Top Sector Relocatable

2. TYP1 through TYP12 specify the relocation mode for DATA (1) through DATA (12), respectively.

TYP	Relocation Mode
0	Absolute
1	Base Sector
2	Top Sector
3	External

3. If 8-bit "data" is specified by the assembler, bits 15 through 8 of DATA (1) through DATA (12) are set to zero, and bits 7 through 0 contain the "data".

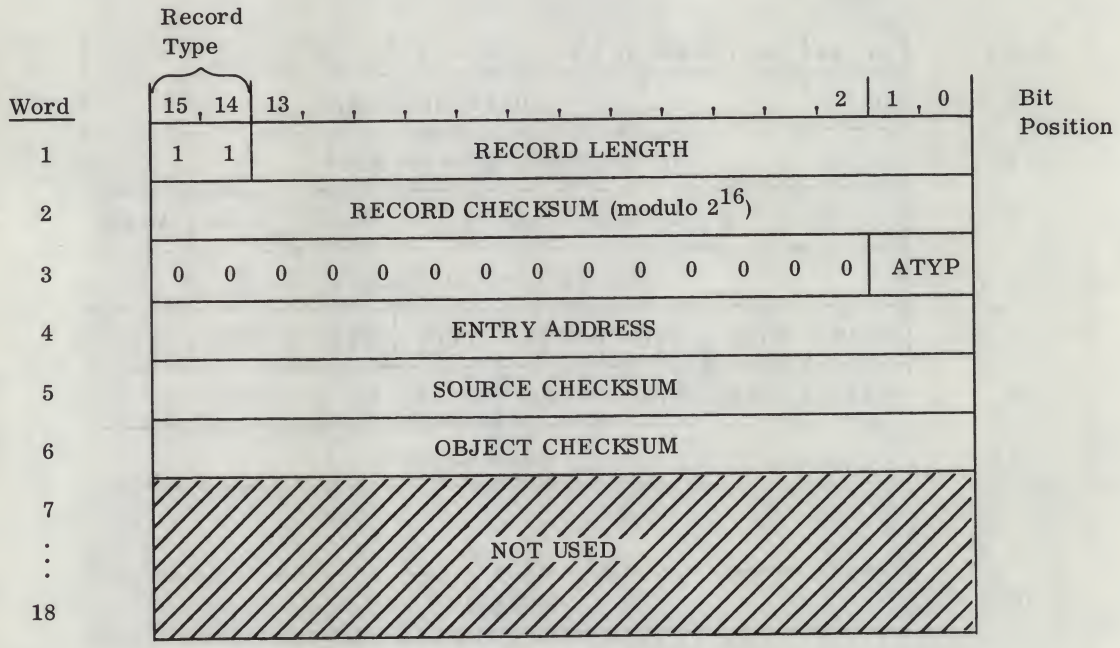
Figure 8-4. Data Record Format

8.1.3.4 End Records

The End Record marks the end of the LM file and specifies an entry address for the load module. The End Record format is illustrated in figure 8-5.

The source checksum represents the sum (modulo-2^{16}) of all the characters, taken one at a time, in the program source file. The source checksum is printed on the program listing following the symbol table printout.

The object checksum represents the modulo-2^{16} sum of all the individual record checksums of the LM. The object checksum also is printed on the program listing following the symbol table.



NOTE: ATYP specifies the mode of the entry address.

ATYP	Address Mode
0	Absolute
1	Base Sector
2	Top Sector
3	External

Figure 8-5. End Record Format

Appendix A

ASCII CHARACTER SET

Table A-1 contains the 7-bit hexadecimal code for each character in the ASCII character set. The printable characters in this set may be set up as program data by use of the .ASCII directive. The remaining characters may be set up in hexadecimal constants with a .WORD directive. Table A-2 contains the legend for nonprintable characters.

Table A-1. ASCII Character Set in Hexadecimal Representation

ASCII Character	7-Bit Hexadecimal Number	Punched Card Code	ASCII Character	7-Bit Hexadecimal Number	Punched Card Code
NUL	00	12-0-1-8-9	0	30	0
SOH	01	12-1-9	1	31	1
STX	02	12-2-9	2	32	2
ETX	03	12-3-9	3	33	3
EOT	04	7-9	4	34	4
ENQ	05	0-5-8-9	5	35	5
ACK	06	0-6-8-9	6	36	6
BEL	07	0-7-8-9	7	37	7
BS	08	11-6-9	8	38	8
HT	09	12-5-9	9	39	9
LF	0A	0-5-9	:	3A	2-8
VT	0B	12-3-8-9	;	3B	11-6-8
FF	0C	12-4-8-9	<	3C	12-4-8
CR	0D	12-5-8-9	=	3D	6-8
SO	0E	12-6-8-9	>	3E	0-6-8
SI	0F	12-7-8-9	?	3F	0-7-8
DLE	10	12-11-1-8-9	@	40	4-8
DC1	11	11-1-9	A	41	12-1
DC2	12	11-2-9	B	42	12-2
DC3	13	11-3-9	C	43	12-3
DC4	14	4-8-9	D	44	12-4
NAK	15	5-8-9	E	45	12-5
SYN	16	2-9	F	46	12-6
ETB	17	0-6-9	G	47	12-7
CAN	18	11-8-9	H	48	12-8
EM	19	11-1-8-9	I	49	12-9
SUB	1A	7-8-9	J	4A	11-1
ESC	1B	0-7-9	K	4B	11-2
FS	1C	11-4-8-9	L	4C	11-3
GS	1D	11-5-8-9	M	4D	11-4
RS	1E	11-6-8-9	N	4E	11-5
US	1F	11-7-8-9	O	4F	11-6
SP	20	No Punches	P	50	11-7
!	21	11-2-8	Q	51	11-8
"	22	7-8	R	52	11-9
#	23	3-8	S	53	0-2
$	24	11-3-8	T	54	0-3
%	25	0-4-8	U	55	0-4
&	26	12	V	56	0-5
'	27	5-8	W	57	0-6
(	28	12-5-8	X	58	0-7
)	29	11-5-8	Y	59	0-8
*	2A	11-4-8	Z	5A	0-9
+	2B	12-6-8	[	5B	12-2-8
,	2C	0-3-8	\	5C	0-8-2
-	2D	11	]	5D	12-7-8
.	2E	12-3-8	↑	5E	11-7-8
/	2F	0-1	←	5F	0-5-8
			`	60	8-1

Table A-1. ASCII Character Set in Hexadecimal Representation (Continued)

ASCII Character	7-Bit Hexadecimal Number	Punched Card Code	ASCII Character	7-Bit Hexadecimal Number	Punched Card Code
a	61	12-0-1	q	71	12-11-8
b	62	12-0-2	r	72	12-11-9
c	63	12-0-3	s	73	11-0-2
d	64	12-0-4	t	74	11-0-3
e	65	12-0-5	u	75	11-0-4
f	66	12-0-6	v	76	11-0-5
g	67	12-0-7	w	77	11-0-6
h	68	12-0-8	x	78	11-0-7
i	69	12-0-9	y	79	11-0-8
j	6A	12-11-1	z	7A	11-0-9
k	6B	12-11-2		7B	12-0
l	6C	12-11-3		7C	12-11
m	6D	12-11-4	ALT	7D	11-0
n	6E	12-11-5	ESC	7E	11-0-1
o	6F	12-11-6	DEL, RUB	7F	12-7-9
p	70	12-11-7			

Table A-2. Legend for Nonprintable Characters

Character	Definition	Character	Definition
NUL	NULL	SI	SHIFT IN
SOH	START OF READING; ALSO START OF MESSAGE	DLE	DATA LINK ESCAPE
		DC1	DEVICE CONTROL 1
STX	START OF TEXT; ALSO EOA, END OF ADDRESS	DC2	DEVICE CONTROL 2
		DC3	DEVICE CONTROL 3
ETX	END OF TEXT; ALSO EOM, END OF MESSAGE	DC4	DEVICE CONTROL 4
		NAK	NEGATIVE ACKNOWLEDGE
EOT	END OF TRANSMISSION (END)	SYN	SYNCHRONOUS IDLE (SYNC)
ENQ	ENQUIRY (ENQRY); ALSO WRU	ETB	END OF TRANSMISSION BLOCK
ACK	ACKNOWLEDGE, ALSO RU		
BEL	RINGS THE BELL	CAN	CANCEL (CANCL)
BS	BACKSPACE	EM	END OF MEDIUM
HT	HORIZONTAL TAB	SUB	SUBSTITUTE
LF	LINE FEED OR LINE SPACE (NEW LINE): ADVANCES PAPER TO NEXT LINE BEGINNING OF LINE	ESC	ESCAPE, PREFIX
		FS	FILE SEPARATOR
		GS	GROUP SEPARATOR
		RS	RECORD SEPARATOR
VT	VERTICAL TAB (VTAB)	US	UNIT SEPARATOR
FF	FORM FEED TO TOP OF NEXT PAGE (PAGE)	SP	SPACE
		ALT ESC	ALT MODE ESCAPE. PREFIX
CR	CARRIAGE RETURN	DEL, RUB	DELETE, RUBOUT
SO	SHIFT OUT		

Appendix B

INDEX OF INSTRUCTIONS

Table B-1. Opcode Index of Instructions

Opcode Base	Mnemonic	Format	Operation	Page
0000	HALT		Halt	4-21
0400	CFR	r	Copy Flags into Register	4-13
0800	CRF	r	Copy Register into Flags	4-13
0C00	PUSHF		Push Flags onto Stack	4-14
1000	PULLF		Pull Flags off Stack	4-14
1400	JSR	disp(xr)	Jump to Subroutine	4-3
1800	JMP	disp(xr)	Jump	4-2
1C00	XCHRS	r	Exchange Register and Stack	4-12
2000	ROL	r, n, *l*	Rotate Left	4-19
2400	ROR	r, n, *l*	Rotate Right	4-20
2800	SHL	r, n, *l*	Shift Right	4-17
2C00	SHR	r, n, *l*	Shift Left	4-18
3000	PFLG	fc	Pulse Flag	4-22
3080	SFLG	fc	Set Flag	4-22
4000	BOC	cc, disp	Branch On Condition	4-1
5000	LI	r, disp	Load Immediate	4-11
5400	RAND	sr, dr	Register AND	4-16
5800	RXOR	sr, dr	Register Exclusive-OR	4-16
5C00	RCPY	sr, dr	Register Copy	4-12
5C00	NOP		No Operation	4-22
6000	PUSH	r	Push Register onto Stack	4-13
6400	PULL	r	Pull Register off Stack	4-14
6800	RADD	sr, dr	Register Add	4-15
6C00	RXCH	sr, dr	Register Exchange	4-12
7000	CAI	r, disp	Complement and Add Immediate	4-17
7400	RADC	sr, dr	Register Add with Carry	4-15
7800	AISZ	r, disp	Add Immediate, Skip if Zero	4-6
7C00	RTI	disp	Return from Interrupt	4-4
8000	RTS	disp	Return from Subroutine	4-4
8800	DECA	0, disp(xr)	Decimal Add	4-11
8C00	ISZ	disp(xr)	Increment and Skip if Zero	4-6
9000	SUBB	0, disp(xr)	Subtract with Borrow	4-10
9400	JSR	@disp(xr)	Jump to Subroutine Indirect	4-3
9800	JMP	@disp(xr)	Jump Indirect	4-3
9C00	SKG	0, disp(xr)	Skip if Greater	4-5
A000	LD	0, @disp(xr)	Load Indirect	4-7
A400	OR	0, disp(xr)	OR	4-10
A800	AND	0, disp(xr)	AND	4-9
AC00	DSZ	disp(xr)	Decrement and Skip if Zero	4-6
B000	ST	0, @disp(xr)	Store Indirect	4-8
B800	SKAZ	0, disp(xr)	Skip if AND is Zero	4-5
BC00	LSEX	0, disp(xr)	Load with Sign Extended	4-8
C000	LD	r, disp(xr)	Load	4-7
D000	ST	r, disp(xr)	Store	4-8
E000	ADD	r, disp(xr)	Add	4-10
F000	SKNE	r, disp(xr)	Skip if Not Equal	4-5

Table B-2. Mnemonic Index of Instructions

Opcode	Mnemonic and Assembler Format		Description	Page
E000	ADD	r,disp(xr)	Add	4-10
7800	AISZ	r,disp	Add Immediate, Skip if Zero	4-6
A800	AND	0,disp(xr)	AND	4-9
4000	BOC	cc,disp	Branch On Condition	4-1
7000	CAI	r,disp	Complement and Add Immediate	4-17
0400	CFR	r	Copy Flags into Register	4-13
0800	CRF	r	Copy Register into Flags	4-13
8800	DECA	0,disp(xr)	Decimal Add	4-11
AC00	DSZ	disp(xr)	Decrement and Skip if Zero	4-6
0000	HALT		Halt	4-21
8C00	ISZ	disp(xr)	Increment and Skip if Zero	4-6
1800	JMP	disp(xr)	Jump	4-2
9800	JMP	@disp(xr)	Jump Indirect	4-3
1400	JSR	disp(xr)	Jump to Subroutine	4-3
9400	JSR	@disp(xr)	Jump to Subroutine Indirect	4-3
C000	LD	r,disp(xr)	Load	4-7
A000	LD	0,@disp(xr)	Load Indirect	4-7
5000	LI	r,disp	Load Immediate	4-11
BC00	LSEX	0,disp(xr)	Load with Sign Extended	4-8
5C00	NOP		No Operation	4-22
A400	OR	0,disp(xr)	OR	4-10
3000	PFLG	fc	Pulse Flag	4-22
6400	PULL	r	Pull Register off Stack	4-14
1000	PULLF		Pull Flags off Stack	4-14
6000	PUSH	r	Push Register onto Stack	4-13
0C00	PUSHF		Push Flags onto Stack	4-14
7400	RADC	sr,dr	Register Add with Carry	4-15
6800	RADD	sr,dr	Register Add	4-15
5400	RAND	sr,dr	Register AND	4-16
5C00	RCPY	sr,dr	Register Copy	4-12
2000	ROL	r,n,*l*	Rotate Left	4-19
2400	ROR	r,n,*l*	Rotate Right	4-20
7C00	RTI	disp	Return from Interrupt	4-4
8000	RTS	disp	Return from Subroutine	4-4
6C00	RXCH	sr,dr	Register Exchange	4-12
5800	RXOR	sr,dr	Register Exclusive-OR	4-16
3080	SFLG	fc	Set Flag	4-22
2800	SHL	r,n,*l*	Shift Left	4-18
2C00	SHR	r,n,*l*	Shift Right	4-17
B800	SKAZ	0,disp(xr)	Skip if AND is Zero	4-5
9C00	SKG	0,disp(xr)	Skip if Greater	4-5
F000	SKNE	r,disp(xr)	Skip if Not Equal	4-5
D000	ST	r,disp(xr)	Store	4-8
B000	ST	0,@disp(xr)	Store Indirect	4-8
9000	SUBB	0,disp(xr)	Subtract with Borrow	4-10
1C00	XCHRS	r	Exchange Register and Stack	4-12

Table B-3. Numeric Index of Instructions

ALPHANUMERIC SEQUENCE BY HEXADECIMAL

Read down then right.

Mnemonic Assembler Code		AC0	AC1	AC2	AC3	BASE PAGE (XX)	PC REL (XX+PC)	AC2 REL (XX+AC2)	AC3 REL (XX+AC3)								
HALT	0000																Halt
CFR r		0400	0500	0600	0700												Copy flags to register
CRF r		0800	0900	0A00	0B00												Copy register to flags
PUSHF	0C00																Push flags onto stack
PULLF	1000																Pull stack into flags
JSR disp(xr)						14XX	15XX	16XX	17XX								Jump to subroutine; XX = ±127; push PC onto stack
JMP disp(xr)						18XX	19XX	1AXX	1BXX								Jump; XX = ±127
XCHRS r		1C00	1D00	1E00	1F00												Exchange register and stack
ROL r,n,l		20XX	21XX	22XX	23XX												Rotate register left
ROR r,n,l		24XX	25XX	26XX	27XX												Rotate register right — Bit 1 = 1 include link bit
SHL r,n,l		28XX	29XX	2AXX	2BXX												Shift left — Bit 2 = 2 shift count
SHR r,n,l		2CXX	2DXX	2EXX	2FXX												Shift right — Bits 2-7 = N = shift count
fc		NOT USED	IE1	IE2	IE3	IE4	IE5	OVF	CRY	LINK	IEN	BYTE	F11	F12	F13	F14	NOT USED	
PFLG fc		3000	3100	3200	3300	3400	3500	3600	3700	3800	3900	3A00	3B00	3C00	3D00	3E00	3F00	Pulse or reset flag
SFLG fc		3080	3180	3280	3380	3480	3580	3680	3780	3880	3980	3A80	3B80	3C80	3D80	3E80	3F80	Set flag
cc		STACK Full	AC0 = 0	AC0 Bit15=0	AC0 Bit0=1	AC0 Bit1=1	AC0 ≠0	AC0 Bit2=1	CONT	LINK	IEN	CRY	AC0 Bit 15=1	OVF	JC13	JC14	JC15	
BOC cc,disp		40XX	41XX	42XX	43XX	44XX	45XX	46XX	47XX	48XX	49XX	4AXX	4BXX	4CXX	4DXX	4EXX	4FXX	Branch on condition (PC relative) XX = ±127
		AC0	AC1	AC2	AC3													
LI r, data		50XX	51XX	52XX	53XX													Load immediate; load register with XX; XX = data Bit 7 of XX extends to Bits 8-15 of register
sr dr		AC0 AC0	AC1 AC0	AC2 AC0	AC3 AC0	AC0 AC1	AC1 AC1	AC2 AC1	AC3 AC1	AC0 AC2	AC1 AC2	AC2 AC2	AC3 AC2	AC0 AC3	AC1 AC3	AC2 AC3	AC3 AC3	
RAND sr,dr		5400	5440	5480	54C0	5500	5540	5580	55C0	5600	5640	5680	56C0	5700	5740	5780	57C0	"AND" register to register; result to register (dr)
RXOR sr,dr		5800	5840	5880	58C0	5900	5940	5980	59C0	5A00	5A40	5A80	5AC0	5B00	5B40	5B80	5BC0	Exclusive "OR" register to register; result to register (dr)
NOP	5C00																	
RCPY sr,dr		5C00	5C40	5C80	5CC0	5D00	5D40	5D80	5DC0	5E00	5E40	5E80	5EC0	5F00	5F40	5F80	5FC0	Copy register to register
		AC0	AC1	AC2	AC3													
PUSH r		6000	6100	6200	6300													Push register onto stack
PULL r		6400	6500	6600	6700													Pull stack into stack
sr dr		AC0 AC0	AC1 AC0	AC2 AC0	AC3 AC0	AC0 AC1	AC1 AC1	AC2 AC1	AC3 AC1	AC0 AC2	AC1 AC2	AC2 AC2	AC3 AC2	AC0 AC3	AC1 AC3	AC2 AC3	AC3 AC3	
RADD sr,dr		6800	6840	6880	68C0	6900	6940	6980	69C0	6A00	6A40	6A80	6AC0	6B00	6B40	6B80	6BC0	Add register to register; result to register (dr), overflow, and carry
RXCH sr,dr		6C00	6C40	6C80	6CC0	6D00	6D40	6D80	6DC0	6E00	6E40	6E80	6EC0	6F00	6F40	6F80	6FC0	Exchange register
		AC0	AC1	AC2	AC3													
CAI r, data		70XX	71XX	72XX	73XX													Complement register and add XX; result to register Bit 7 of XX is extended to Bits 8-15
sr dr		AC0 AC0	AC1 AC0	AC2 AC0	AC3 AC0	AC0 AC1	AC1 AC1	AC2 AC1	AC3 AC1	AC0 AC2	AC1 AC2	AC2 AC2	AC3 AC2	AC0 AC3	AC1 AC3	AC2 AC3	AC3 AC3	
RADC sr,dr		7400	7440	7480	74C0	7500	7540	7580	75C0	7600	7640	7680	76C0	7700	7740	7780	77C0	Add register to register plus carry; result to register (dr); overflow and carry

ALPHANUMERIC SEQUENCE BY HEXADECIMAL
Read down then right.

Mnemonic	Assembler Code		AC0	AC1	AC2	AC3	BASE PAGE XX	PC REL (XX+PC)	AC2 REL (XX+AC2)	AC3 REL (XX+AC3)	
AISZ	r, data		78XX	79XX	7AXX	7BXX					Add XX to register; skip next instruction if result = zero; XX = ±127
RTI	disp	7CXX									Return from interrupt; add XX to top of stack and place result in PC; XX = ±127; set IEN flag
RTS	disp	80XX									Return from subroutine; add XX to top of stack and place result in PC; XX = ±127
DECA	0, disp(xr)						88XX	89XX	8AXX	8BXX	Decimal add register AC0 to contents of effective address; result to AC0, overflow and carry; address = (XX + register shown); XX = ±127
ISZ	disp(xr)						8CXX	8DXX	8EXX	8FXX	Increment contents of effective address by 1; skip next instruction if result = 0; result is in EA; use address mode shown; XX = ±127
SUBB	0, disp(xr)						90XX	91XX	92XX	93XX	Subtract contents of effective address from AC0; result to AC0; use address mode shown; XX = ±127
JSR	@ disp(xr)						94XX	95XX	96XX	97XX	Jump to subroutine indirect; push PC onto stack; final address = to contents of location (XX + register shown); XX = ±127
JMP	@ disp(xr)						98XX	99XX	9AXX	9BXX	Jump indirect; final address = to contents of location (XX + register shown); XX = ±127
SKG	0, disp(xr)						9CXX	9DXX	9EXX	9FXX	Compare AC0 with contents of location (XX + register shown); XX = ±127; skip next instruction if AC0 > (EA)
LD	0, @ disp(xr)						A0XX	A1XX	A2XX	A3XX	Load indirect; load AC0 with contents of final address; address = contents of location (XX + register shown); XX = ±127
OR	0, disp(xr)						A4XX	A5XX	A6XX	A7XX	OR AC0 with contents of location (XX + register shown); XX = ±127; result to AC0
AND	0, disp(xr)						A8XX	A9XX	AAXX	ABXX	AND AC0 with contents of location (XX + register shown); XX = ±127; result to AC0
DSZ	disp(xr)						ACXX	ADXX	AEXX	AFXX	Decrement contents of effective address by 1; skip next instruction if result = 0; result is in EA; address = (XX + register shown); XX = ±127
ST	0, @ disp(xr)						B0XX	B1XX	B2XX	B3XX	Store indirect; store AC0 into final address; address = contents of location (XX + register shown); XX = ±127
SKAZ	0, disp(xr)						B8XX	B9XX	BAXX	BBXX	AND AC0 with contents of location (XX + register shown); skip next instruction if result = 0; XX = ±127
LSEX	0, disp(xr)						BCXX	BDXX	BEXX	BFXX	Load AC0 with sign extended; Bit 7 of location (XX + register shown) is extended to AC0 8-15; Bits 0-7 are loaded to AC0 Bits 0-7; XX = ±127
LD	r, disp(xr)		X				C0XX	C1XX	C2XX	C3XX	Load AC0 with contents of location (XX + register shown); XX = ±127
				X			C4XX	C5XX	C6XX	C7XX	Load AC1 with contents of location (XX + register shown); XX = ±127
					X		C8XX	C9XX	CAXX	CBXX	Load AC2 with contents of location (XX + register shown); XX = ±127
						X	CCXX	CDXX	CEXX	CFXX	Load AC3 with contents of location (XX + register shown); XX = ±127
ST	r, disp(xr)		X				D0XX	D1XX	D2XX	D3XX	Store AC0 to location (XX + register shown); XX = ±127
				X			D4XX	D5XX	D6XX	D7XX	Store AC1 to location (XX + register shown); XX = ±127
					X		D8XX	D9XX	DAXX	DBXX	Store AC2 to location (XX + register shown); XX = ±127
						X	DCXX	DDXX	DEXX	DFXX	Store AC3 to location (XX + register shown); XX = ±127
ADD	r, disp(xr)		X				E0XX	E1XX	E2XX	E3XX	Add AC0 to location (XX + register shown); XX = ±127; result to AC0
				X			E4XX	E5XX	E6XX	E7XX	Add AC1 to location (XX + register shown); XX = ±127; result to AC1
					X		E8XX	E9XX	EAXX	EBXX	Add AC2 to location (XX + register shown); XX = ±127; result to AC2
						X	ECXX	EDXX	EEXX	EFXX	Add AC3 to location (XX + register shown); XX = ±127; result to AC3
SKNE	r, disp(xr)		X				F0XX	F1XX	F2XX	F3XX	Compare AC0 to location (XX + register shown); XX = ±127; if not equal skip next instruction
				X			F4XX	F5XX	F6XX	F7XX	Compare AC1 to location (XX + register shown); XX = ±127; if not equal skip next instruction
					X		F8XX	F9XX	FAXX	FBXX	Compare AC2 to location (XX + register shown); XX = ±127; if not equal skip next instruction
						X	FCXX	FDXX	FEXX	FFXX	Compare AC3 to location (XX + register shown); XX = ±127; if not equal skip next instruction

Appendix C

INSTRUCTION EXECUTION TIMES

The formats for computing the execution times of PACE instructions are listed in table C-1. The formulas are presented in terms of machine (microinstruction) cycles (M) and I/O data-transfer cycle extends (E_R for read and E_W for write). Each machine cycle (M) consists of four clock cycles. The following example shows the method of calculating the instruction execution times.

Mnemonic	Instruction Execution Time	
	Formula	μsecs (typical excluding E_R and E_W)
ADD	$4M + 2E_R$	12
AISZ	$5M + E_R$ + (1M if skip)	15, 18
AND	$4M + 2E_R$	12
BOC	$5M + E_R$ + (1M if skip)	15, 18
CAI	$5M + E_R$	15
CFR	$4M + E_R$	12
CRF	$4M + E_R$	12
DECA	$7M + 2E_R$	21
DSZ	$7M + 2E_R + E_W$ + (1M if skip)	21, 24
HALT	——	—
ISZ	$7M + 2E_R + E_W$ + (1M if skip)	21, 24
JMP	$4M + E_R$	12
JMP@	$4M + 2E_R$	12
JSR	$5M + E_R$	15
JSR@	$5M + 2E_R$	15
LD	$4M + 2E_R$	12
LD@	$5M + 3E_R$	15
LI	$4M + E_R$	12
LSEX	$4M + 2E_R$	12
NOP	$4M + E_R$	12
OR	$4M + E_R$	12
PFLG	$6M + E_R$	18
PULL	$4M + E_R$	12
PULLF	$4M + E_R$	12
PUSH	$4M + E_R$	15, 18
PUSHF	$4M + E_R$	12
RADC	$4M + E_R$	12
RADD	$4M + E_R$	12
RAND	$4M + E_R$	12
RCPY	$4M + E_R$	12
ROL	$(5 + 3n)M + E_R$, n = 1 to 127; $6M + E_R$, n = 0	18 to 1158
ROR	$(5 + 3n)M + E_R$, n = 1 to 127; $6M + E_R$, n = 0	18 to 1158
RTI	$6M + E_R$	18
RTS	$5M + E_R$	15
RXCH	$6M + E_R$	18
RXOR	$4M + E_R$	12
SFLG	$5M + E_R$	15
SHL	$(5 + 3n)M + E_R$, n = 1 to 127; $6M + E_R$, n = 0	18 to 1158
SHR	$(5 + 3n)M + E_R$, n = 1 to 127; $6M + E_R$, n = 0	18 to 1158
SKAZ	$5M + 2E_R$ + (1M if skip)	15, 18
SKG	$7M + 2E_R$ + (1M if skip)	21, 24
SKNE	$5M + 2E_R$ + (1M if skip)	15, 18
ST	$4M + E_R + E_W$	12
ST@	$4M + 2E_R + E_W$	12
SUBB	$4M + 2E_R$	12
XCHRS	$6M + E_R$	18

NOTE: M = machine cycle time = 4 clock periods = 3 μsecs (typically)
E_R = Extended time for read cycles } Required only if memory access > 1.5 μsec.
E_W = Extended time for write cycles } Typical E_R and E_W values are 750 nsec.
n = number of shifts, External interrupt response time is $7M + E_R$ plus time to finish current instruction

Appendix D

DIRECTIVES

Directive	Mnemonic	Operands
Title	.TITLE	symbol [,string]
Split	.SPLIT	
End	.END	[address]
Program Section	.ASECT	expression
	.BSECT	expression
	.TSECT	expression
List	.LIST	immediate
Space	.SPACE	immediate
Page	.PAGE	[string]
Word	.WORD	expression [,expression...]
ASCII	.ASCII	string [,string...]
Set	.SET	symbol,expression
Conditional Assembly	.IF	expression
	.ELSE	
	.ENDIF	
Global	.GLOBL	symbol [,symbol...]
Local	.LOCAL	
Data Length	.DLEN	expression
Pointer	.PTR	expression [,expression...]
Pool	.POOL	expression
No Base	.NOBAS	
Macro *	.MACRO	mname [parameters]
Macro End *	.ENDM	
Macro Local *	.MLOC	symbol [,symbol]
Do *	.DO	count
End Do *	.ENDDO	
Exit Do *	.EXIT	
Conditional Assembly	.IFC	string1 operator string2
Error	.ERROR	string
Macro Delete	.MDEL	mname [,mname...]

* Cannot be used outside a macro definition.

NOTE: In the .IFC directive, operator means a relational operator and may be either "EQ" for equal for "NE" for not equal.

Appendix E

PROGRAMMERS CHECKLIST

The following list of items is suggested for desk-checking a program prior to assembly.

1. Is the source program terminated by an . END Directive?
2. Is each label in the program terminated by a colon (:)?
3. Is each comment in the program preceded by a semicolon (;)?
4. Is each string constant in the program set off on both ends by a prime (')?
5. Are all external symbols listed in . GLOBL Statements?
6. Are all hexadecimal constants preceded by either X' or 0 (zero)?
7. For each . IF Directive in the program, is there a corresponding . ENDIF?
8. Are any global symbols defined by forward references? Such definition is illegal.
9. Are any symbols defined by two-level forward references? Such definition is illegal.
10. For each . MACRO directive in the program, is there a corresponding . ENDM?
11. For each . DO directive in the program, is there a corresponding . ENDDO?

Appendix F

POSITIVE POWERS OF TWO

n	2^n
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024
11	2048
12	4096
13	8192
14	16384
15	32768
16	65536
17	13107 2
18	26214 4
19	52428 8
20	10485 76
21	20971 52
22	41943 04
23	83886 08
24	16777 216
25	33554 432
26	67108 864
27	13421 7728
28	26843 5456
29	53687 0912
30	10737 41824
31	21474 83648
32	42949 67296
33	85899 34592
34	17179 86918 4
35	34359 73836 8
36	68719 47673 6
37	13743 89534 72
38	27487 79069 44
39	54975 58138 88
40	10995 11627 776
41	21990 23255 552
42	43980 46511 104
43	87960 93022 208
44	17592 18604 4416
45	35184 37208 8832
46	70368 74417 7664
47	14073 74883 55328
48	28147 49767 10656
49	56294 99534 21312
50	11258 99906 84262 4
51	22517 99813 68524 8
52	45035 99627 37049 6
53	90071 99254 74099 2
54	18014 39850 94819 84
55	36028 79701 89639 68
56	72057 59403 79279 36
57	14411 51880 75855 872
58	28823 03761 51711 744
59	57646 07523 03423 488
60	11529 21504 60684 6976
61	23058 43009 21369 3952
62	46116 86018 42738 7904
63	92233 72036 85477 5808
64	18446 74407 37095 51616
65	36893 48814 74191 03232
66	73786 97629 48382 06464
67	14757 39525 89676 41292 8
68	29514 79051 79352 82585 6
69	59029 58103 58705 65171 2
70	11805 91620 71741 13034 24
71	23611 83241 43482 26068 48
72	47223 66482 86964 52136 96
73	94447 32965 73929 04273 92
74	18889 46593 14785 80854 784
75	37778 93186 29571 61709 568
76	75557 86372 59143 23419 136
77	15111 57274 51828 64683 8272
78	30223 14549 03657 29367 6544
79	60446 29098 07314 58735 3088
80	12089 25819 61462 91747 06176
81	24178 51639 22925 83494 12352
82	48357 03278 45851 66988 24704
83	96714 06556 91703 33976 49408
84	19342 81311 38340 66795 29881 6
85	38685 62622 76681 33590 59763 2
86	77371 25245 53362 67181 19526 4
87	15474 25049 10672 53436 23905 28
88	30948 50098 21345 06872 47810 56
89	61897 00196 42690 13744 95621 12
90	12379 40039 28538 02748 99124 224
91	24758 80078 57076 05497 98248 448
92	49517 60157 14152 10995 96496 896
93	99035 20314 28304 21991 92993 792
94	19807 04062 85660 84398 38598 7584
95	39614 08125 71321 68796 77197 5168
96	79228 16251 42643 37593 54395 0336
97	15845 63250 28528 67518 70879 00672
98	31691 26500 57057 35037 41758 01344
99	63382 53001 14114 70074 83516 02688
100	12676 50600 22822 94014 96703 20537 6
101	25353 01200 45645 88029 93406 41075 2

Appendix G

NEGATIVE POWERS OF TWO

n	2^{-n}
0	1.0
1	0.5
2	0.25
3	0.125
4	0.0625
5	0.03125
6	0.01562 5
7	0.00781 25
8	0.00390 625
9	0.00195 3125
10	0.00097 65625
11	0.00048 82812 5
12	0.00024 41406 25
13	0.00012 20703 125
14	0.00006 10351 5625
15	0.00003 05175 78125
16	0.00001 52587 89062 5
17	0.00000 76293 94531 25
18	0.00000 38146 97265 625
19	0.00000 19073 48632 8125
20	0.00000 09536 74316 40625
21	0.00000 04768 37158 20312 5
22	0.00000 02384 18579 10156 25
23	0.00000 01192 09289 55078 125
24	0.00000 00596 04644 77539 0625
25	0.00000 00298 02322 38769 53125
26	0.00000 00149 01161 19384 76562 5
27	0.00000 00074 50580 59692 38281 25
28	0.00000 00037 25290 29846 19140 625
29	0.00000 00018 62645 14923 09570 3125
30	0.00000 00009 31322 57461 54785 15625
31	0.00000 00004 65661 28730 77392 57812 5
32	0.00000 00002 32830 64365 38696 28906 25
33	0.00000 00001 16415 32182 69348 14453 125
34	0.00000 00000 58207 66091 34674 07226 5625
35	0.00000 00000 29103 83045 67337 03613 28125
36	0.00000 00000 14551 91522 83668 51806 64062 5
37	0.00000 00000 07275 95761 41834 25903 32031 25
38	0.00000 00000 03637 97880 70917 12951 66015 625
39	0.00000 00000 01818 98940 35458 56475 83007 8125
40	0.00000 00000 00909 49470 17729 28237 91503 90625
41	0.00000 00000 00454 74735 08864 64118 95751 95312 5
42	0.00000 00000 00227 37367 54432 32059 47875 97656 25
43	0.00000 00000 00113 68683 77216 16029 73937 98828 125
44	0.00000 00000 00056 84341 88608 08014 86968 99414 0625
45	0.00000 00000 00028 42170 94304 04007 43484 49707 03125
46	0.00000 00000 00014 21085 47152 02003 71742 24853 51562 5
47	0.00000 00000 00007 10542 73576 01001 85871 12426 75781 25
48	0.00000 00000 00003 55271 36788 00500 92935 56213 37890 625
49	0.00000 00000 00001 77635 68394 00250 46467 78106 68945 3125
50	0.00000 00000 00000 88817 84197 00125 23233 89053 34472 65625

Appendix H

THE HEXADECIMAL NUMBER SYSTEM

We have been taught from childhood to recognize and manipulate a number system called decimal or base-10, which uses ten symbols to represent values or numbers. These symbols are 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. Combinations of these form other numbers, and each number or digit position is assigned a value equal to its position in the number sequence. For example, the number 12,045:

POSITION NO.	4	3	2	1	0	
	1	2	0	4	5	
					$= 5 \times 10^0 =$	5
					$= 4 \times 10^1 =$	40
					$= 0 \times 10^2 =$	000
					$= 2 \times 10^3 =$	2,000
					$= 1 \times 10^4 =$	10,000
						$12{,}045_{10}$

10 is the base-value of the number system, and 0, 1, 2, 3, 4 are the positions of weighted values.

Most computers use a base-2 numbering system in which zeros and ones are the only symbols used to represent any number. The least-significant bit would have a value of 2^0, the next bit would be 2^1, then 2^2, etc. Let's use a group of five bits and assign bit 0 as the least significant bit.

BIT NO.						
0		1		1×2^0		1
1		0		0×2^1		0
2	=	1	=	1×2^2	=	4
3		0		0×2^3		0
4		1		1×2^4		16
						21_{10}

21 is the sum of the values of the bit positions.

It can also be seen that by using larger groups of bits, larger numbers may be represented. An eight-bit computer, which can handle eight bit positions in parallel, can represent numbers from 0 to 255_{10}.

All Bits Equal 0

BIT NO.						
0		0		0×2^0		0
1		0		0×2^1		0
2		0		0×2^2		0
3	=	0	=	0×2^3	=	0
4		0		0×2^4		0
5		0		0×2^5		0
6		0		0×2^6		0
7		0		0×2^7		0
						0_{10}

All Bits Equal 1

BIT NO.						
0		1		1×2^0		1
1		1		1×2^1		2
2		1		1×2^2		4
3	=	1		1×2^3	=	8
4		1		1×2^4		16
5		1		1×2^5		32
6		1		1×2^6		64
7		1		1×2^7		128
						255_{10}

A computer that has 16 bit positions may represent numbers with values from zero to 65,535.

Another consideration in computers is the representation of both positive and negative values. In the "sign magnitude" system, this may be accomplished by assigning one of the bits in a group as a plus/minus indicator. The normal method is to assign the most-significant bit position to this task. If it is a logic zero, then the value is positive; if it is a logic one, then the value is minus. Assuming a group of eight bits maximum, and using the eighth position as the sign, we may represent the following numbers:

BIT NO.						
0		1		1×2^0		1
1		1		1×2^1		2
2		1		1×2^2		4
3	=	1	=	1×2^3	=	8
4		1		1×2^4		16
5		1		1×2^5		32
6		1		1×2^6		64
sign bit 7		0	=	+		$+127_{10}$

If bit 7 is equal to a 1, then the above number would be negative: −127. Note that by using the most-significant bit for the sign, the maximum number that may be represented is only ±127. In a 16-bit computer this number would be ±32,767.

Because it is difficult for us to convert visually many ones and zeros to their represented value, other methods of representing numbers have been implemented.

BCD OR BINARY CODED DECIMAL:

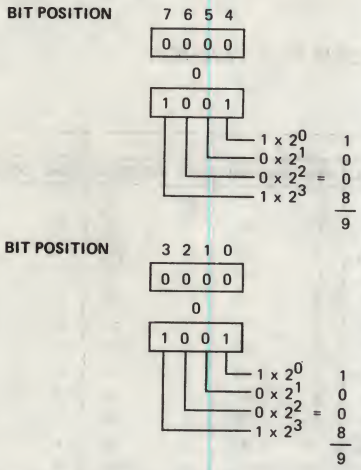
BCD uses groups of four binary bits or positions, and only uses those combinations that add up to 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9. For example:

BIT	3	2	1	0	
	0	0	0	0	= 0
	0	0	0	1	= 1
	0	0	1	0	= 2
	0	0	1	1	= 3
	0	1	0	0	= 4
	0	1	0	1	= 5
	0	1	1	0	= 6
	0	1	1	1	= 7
	1	0	0	0	= 8
	1	0	0	1	= 9

The other binary combinations possible in the four bit positions are not allowed in the BCD method:

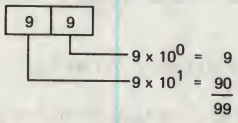
1	0	1	0	Not Valid
1	0	1	1	
1	1	0	0	
1	1	0	1	
1	1	1	0	
1	1	1	1	

In an 8-bit computer, the decimal numbers 00 through 99 may be represented:



Note that the binary weighting system repeats for each four-bit group.

This is then compensated for by applying the decimal (base-10) rules to the converted numbers:



(By having to weigh only up to four binary bits, you quickly become efficient at converting binary numbers to decimal form and decimal numbers to binary form.)

The maximum numbers that can be represented in an 8-bit machine is then only 99_{10} in decimal versus 255_{10} in binary.

As you can see, the efficiency of a computer is restricted because of the illegal combination in each four-bit group. Another representation of binary numbers allows for *all* combinations of the four-bit groups. This system is called hexadecimal representation.

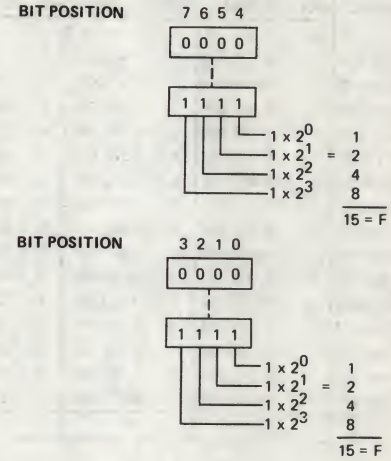
HEXADECIMAL (HEX) NOTATION

Hex uses a numbering system of base 16, and allows for all combinations of the four-bit binary groups, as follows:

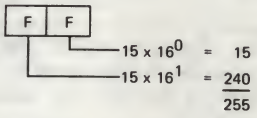
BIT POSITION: 3 2 1 0	BINARY	HEX SYMBOL
0 0 0 0	0	0
0 0 0 1	1	1
0 0 1 0	2	2
0 0 1 1	3	3
0 1 0 0	4	4
0 1 0 1	5	5
0 1 1 0	6	6
0 1 1 1	7	7
1 0 0 0	8	8
1 0 0 1	9	9
1 0 1 0	10	A
1 0 1 1	11	B
1 1 0 0	12	C
1 1 0 1	13	D
1 1 1 0	14	E
1 1 1 1	15	F

The notations A through F are used to allow for a single-character representation of the four-bit group without duplication.

With hex we can now represent all 16 combinations of binary weights possible in a group of four bit positions. An eight bit computer can then represent the numbers 00 through FF, which is equivalent to binary 0 through 255:



Applying the same rules as for decimal, but using the base 16 instead of base 10:



Thus, binary numbers, no matter what the number of position, can easily be converted simply by dividing them up into groups of four bits. For example, in a 16-bit computer:

Hex	F	E	9	A
	∧	∧	∧	∧
Binary	1111	1110	1001	1010
	∨	∨	∨	∨
Hex	F	E	9	A

Further, the use of hex symbols as an equivalent for four binary bits requires fewer printed symbols, and most computer documentation today uses the hexadecimal code representation.

POSITIVE AND NEGATIVE NUMBERS:

In hex or in binary, the method of representing positive and negative numbers is the same. The most-significant bit of the most-significant group is set to a zero for a positive number or a one for a negative number.

If there are four groups of 4-bits each, as in a 16-bit computer, we could have:

Hex	7	F	F	F
	∧	∧	∧	∧
Binary	0111	1111	1111	1111
	sign bit			

This number is equivalent to +32,767.

Appendix I

HEXADECIMAL AND DECIMAL INTEGER CONVERSION

8		7		6		5		4		3		2		1	
HEX	DECIMAL	HEX	DECIMAL	HEX	DECIMAL	HEX	DECIMAL	HEX	DECIMAL	HEX	DECIMAL	HEX	DECIMAL	HEX	DECIMAL
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	268 435 456	1	16 777 216	1	1 048 576	1	65 536	1	4 096	1	256	1	16	1	1
2	536 870 912	2	33 554 432	2	2 097 152	2	131 072	2	8 192	2	512	2	32	2	2
3	805 306 368	3	50 331 648	3	3 145 728	3	196 608	3	12 288	3	768	3	48	3	3
4	1 073 741 824	4	67 108 864	4	4 194 304	4	262 144	4	16 384	4	1 024	4	64	4	4
5	1 342 177 280	5	83 886 080	5	5 242 880	5	327 680	5	20 480	5	1 280	5	80	5	5
6	1 610 612 736	6	100 663 296	6	6 291 456	6	393 216	6	24 576	6	1 536	6	96	6	6
7	1 879 048 192	7	117 440 512	7	7 340 032	7	458 752	7	28 672	7	1 792	7	112	7	7
8	2 147 483 648	8	134 217 728	8	8 388 608	8	524 288	8	32 768	8	2 048	8	128	8	8
9	2 415 919 104	9	150 994 944	9	9 437 184	9	589 824	9	36 864	9	2 304	9	144	9	9
A	2 684 354 560	A	167 772 160	A	10 485 760	A	655 360	A	40 960	A	2 560	A	160	A	10
B	2 952 790 016	B	184 549 376	B	11 534 336	B	720 896	B	45 056	B	2 816	B	176	B	11
C	3 221 225 472	C	201 326 592	C	12 582 912	C	786 432	C	49 152	C	3 072	C	192	C	12
D	3 489 660 928	D	218 103 808	D	13 631 488	D	851 968	D	53 248	D	3 328	D	208	D	13
E	3 758 096 384	E	234 881 024	E	14 680 064	E	917 504	E	57 344	E	3 584	E	224	E	14
F	4 026 531 840	F	251 658 240	F	15 728 640	F	983 040	F	61 440	F	3 840	F	240	F	15
8		7		6		5		4		3		2		1	

TO CONVERT HEXADECIMAL TO DECIMAL

1. Locate the column of decimal numbers corresponding to the left-most digit or letter of the hexadecimal; select from this column and record the number that corresponds to the position of the hexadecimal digit or letter.
2. Repeat step 1 for the next (second from the left) position.
3. Repeat step 1 for the units (third from the left) position.
4. Add the numbers selected from the table to form the decimal number.

TO CONVERT DECIMAL TO HEXADECIMAL

1. (a) Select from the table the highest decimal number that is equal to or less than the number to be converted.
   (b) Record the hexadecimal of the column containing the selected number.
   (c) Subtract the selected decimal from the number to be converted.
2. Using the remainder from step 1(c) repeat all of step 1 to develop the second position of the hexadecimal (and a remainder).
3. Using the remainder from step 2 repeat all of step 1 to develop the units position of the hexadecimal.
4. Combine terms to form the hexadecimal number.

To convert integer numbers greater than the capacity of table, use the techniques below:

HEXADECIMAL TO DECIMAL

Successive cumulative multiplication from left to right, adding units position.

Example: $D34_{16} = 3380_{10}$

```
D =   13
    x16
    208
3 =  +3
    211
    x16
   3376
4 =  +4
   3380
```

EXAMPLE

Conversion of Hexadecimal Value	D34
D	3328
3	48
4	4
Decimal	3380

DECIMAL TO HEXADECIMAL

Divide and collect the remainder in reverse order.

Example: $3380_{10} = D34_{16}$

```
16 | 3380 ---- 4   remainder
16 |  211 ---- 3        ^
16 |   13 ---- D        |
```

EXAMPLE

Conversion of Decimal Value	3380
D	−3328
	52
3	−48
	4
4	−4
Hexadecimal	D34

Appendix J

HEXADECIMAL AND DECIMAL FRACTION CONVERSION

1		2		3		4	
HEX	DECIMAL	HEX	DECIMAL	HEX	DECIMAL	HEX	DECIMAL EQUIVALENT
.0	.0000	.00	.0000 0000	.000	.0000 0000 0000	.0000	.0000 0000 0000 0000
.1	.0625	.01	.0039 0625	.001	.0002 4414 0625	.0001	.0000 1525 8789 0625
.2	.1250	.02	.0078 1250	.002	.0004 8828 1250	.0002	.0000 3051 7578 1250
.3	.1875	.03	.0117 1875	.003	.0007 3242 1875	.0003	.0000 4577 6367 1875
.4	.2500	.04	.0156 2500	.004	.0009 7656 2500	.0004	.0000 6103 5156 2500
.5	.3125	.05	.0195 3125	.005	.0012 2070 3125	.0005	.0000 7629 3945 3125
.6	.3750	.06	.0234 3750	.006	.0014 6484 3750	.0006	.0000 9155 2734 3750
.7	.4375	.07	.0273 4375	.007	.0017 0898 4375	.0007	.0001 0681 1523 4375
.8	.5000	.08	.0312 5000	.008	.0019 5312 5000	.0008	.0001 2207 0312 5000
.9	.5625	.09	.0351 5625	.009	.0021 9726 5625	.0009	.0001 3732 9101 5625
.A	.6250	.0A	.0390 6250	.00A	.0024 4140 6250	.000A	.0001 5258 7890 6250
.B	.6875	.0B	.0429 6875	.00B	.0026 8554 6875	.000B	.0001 6784 6679 6875
.C	.7500	.0C	.0468 7500	.00C	.0029 2968 7500	.000C	.0001 8310 5468 7500
.D	.8125	.0D	.0507 8125	.00D	.0031 7382 8125	.000D	.0001 9836 4257 8125
.E	.8750	.0E	.0546 8750	.00E	.0034 1796 8750	.000E	.0002 1362 3046 8750
.F	.9375	.0F	.0585 9375	.00F	.0036 6210 9375	.000F	.0002 2888 1835 9375
1		2		3		4	

TO CONVERT .ABC HEXADECIMAL TO DECIMAL

Find .A in position 1 .6250
Find .0B in position 2 .0429 6875
Find .00C in position 3 .0029 2968 7500
.ABC Hex is equal to .6708 9843 7500

By making the most-significant-bit a logic 1, then the number becomes:

```
F    F    F    F
^    ^    ^    ^
1111 1111 1111 1111
|
└── sign bit
```

This number is equivalent to −32,767.

The method used to represent a negative hexadecimal number depends on the type of numbering system chosen for binary arithmetic processing. Most digital computers use either the "sign magnitude" system or the twos-complement system. In the sign magnitude system, a negative value is formed by setting a sign bit—the most-significant bit of the most-significant group of bits—to one, and the remaining bits to the desired absolute value. Thus, −32,767 is represented as 1111 1111 1111 1111.

Conversely, if the most-significant-bit is a zero the number is positive; +32,767 is represented as 0111 1111 1111 1111.

In the twos-complement system—the system used in PACE—positive numbers are represented exactly as in the sign magnitude system (sign bit is a logic zero); but negative numbers are represented by the twos-complement of the absolute value of the number. Thus, −32,767 becomes, in the twos-complement system, 1000 0000 0000 0001. Appendix E shows how this conversion is accomplished.

Appendix K

NEGATIVE HEXADECIMAL NUMBERS

The PACE microprocessor maintains negative numbers in twos-complement form. To convert a number in hexadecimal notation to its twos-complement equivalent, subtract the number from hexadecimal 2^n, where "n" is the number of binary bits in the computer word. For a 16-bit word, "n" is 16, and 2^n is 1 0000 0000 0000 0000 (binary) or 1 0000 (hex).

Thus, the negative of 1245_{16} is:

```
 10000
-1245
 EDBB
```

A hexadecimal number will be negative in the PACE CPU if the left-most digit is 8, 9, A, B, C, D, E, or F (because all of these groupings start with a one). Thus, the twos-complement of hex FACE is:

```
 10000
-FACE
+0532
```

Perhaps an easier way to find the twos-complement of a hexadecimal number is first to take the ones-complement of the number; the ones-complement plus one is the twos-complement. The ones-complement of a number is its inverted form; simply exchange its ones for zeros, and its zeros for ones. Thus,

```
hexadecimal     binary equivalent          ones-complement
   FACE  →  1111 1010 1100 1110  →  0000 0101 0011 0001
```

```
                                  ones-complement +1
                                  0000 0101 0011 0001
                                                   +1
                                  ---------------------
                                  0000 0101 0011 0010
                                  ---------------------
Hex twos-complement of FACE  →      0    5    3    2
```

DOCUMENT REVIEW FORM

Your comments concerning this document help us produce better documentation for you.

GENERAL COMMENTS

	Yes	No		Yes	No
Easy to Read?	☐	☐	Complete?	☐	☐
Well Organized?	☐	☐	Well Illustrated?	☐	☐
Accurate?	☐	☐	Suitable for Your Needs?	☐	☐

How do You Use this Document?

- ☐ As an introduction to the subject
- ☐ For additional knowledge
- ☐ For continual reference
- ☐ Other

SPECIFIC CLARIFICATIONS AND/OR CORRECTIONS

Reference	Page No.
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____

This form should not be used as an order blank. Requests for copies of publications should be directed to the National Semiconductor sales office serving your locality.

Send comments to:

National Semiconductor Corporation
2900 Semiconductor Drive
Santa Clara, California 95051
Attention: Microprocessor Publications

FOLD HERE

National Semiconductor Corporation
2900 Semiconductor Drive
Santa Clara, California 95051
(408) 737-5000
TWX: 910-339-9240

National Semiconductor (UK) Ltd.
Larkfield Industrial Estate
Greenock, Scotland
Telephone: GOUROCK 33251
Telex: 778 632

National Semiconductor GmbH
808 Fuerstenfeldbruck
Industriestrasse 10
West Germany
Telephone: (08141) 1371
Telex: 05-27649

NS Electronics (PTE) Ltd.
No. 1100 Lower Delta Rd.
Singapore 3
Telephone: 630011
Telex: NATSEMI RS 21402

NS Electronics SDN BHD
Batu Berendam
Free Trade Zone
Malacca, Malaysia
Telephone: 5171
Telex: NSELECT 519 MALACCA
(c/o Kuala Lumpur)

PUBLICATION NO. 4200130A, ORDER NO. IPC-16S/969Y

PRINTED IN U.S.A. DD10M17